BASIC-80
REFERENCE MANUAL

Manual Order No. 9800758-02

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

The following are trademarks of Intel Corporation and may be used only to describe Intel products:

i	iSBC	Multimodule
ICE	Library Manager	PROMPT
iCS	MCS	Promware
Insite	Megachassis	RMX
Intel	Micromap	UPI
Intelevision	Multibus	μScope
Intellec		

and the combination of ICE, iCS, iSBC, MCS, or RMX and a numerical suffix.

PREFACE

This manual describes and explains the features and conventions of Intel Disk Extended BASIC-80, as implemented on Intel Intellec microcomputer development systems using the Intel Systems Implementation Supervisor (ISIS-II), and on Intel Single Board Computer Systems using the Intel Real-Time Multitasking Executive (RMX-80).

This manual is written for users who require concise, complete information about Intel BASIC-80 characteristics, and organizes this information into seven chapters and five appendices;

"Introduction" describes the general capabilities of BASIC-80, and its operating environment.

"Language Elements" describes the ways BASIC-80 represents its instructions, constants, variables, arrays, operators, and expressions.

"Entering and Editing Programs" shows how you enter text and edit it, at time of entry or after storage.

"Error Handling" shows how errors are identified, trapped, and used to initiate error-resolving routines.

"Disk File Input/Output" describes and shows how random and sequential data files are created and used.

"Commands and Statements" describes each command and statement in alphabetic order.

"Functions" describes each function in alphabetic order.

"Appendix A: BASIC-80 Error Codes" lists all BASIC-80 error messages, descriptions, and codes in tabular format.

"Appendix B: BASIC-80 Reserved Words" lists words that cannot be used in variable names.

"Appendix C: BASIC-80 Command Characters" lists BASIC-80 one-character editing and control characters and their meanings.

"Appendix D: ASCII Codes" lists ASCII codes and their meanings.

"Appendix E: Calling Non-BASIC-80 Subroutines" shows how to prepare and call PL/M-80, FORTRAN-80, and 8080/8085 assembly language subroutines.

"Appendix F: Configuring RMX-80 BASIC-80" shows how to configure BASIC-80 with various hardware systems using RMX-80.

Other Relevant Intel Publications:

The following manuals are required to use BASIC-80 with ISIS-II or RMX-80:

- *ISIS-II User's Guide,* 9800306, which describes how to operate the Intel Systems Implementation Supervisor operating system (ISIS-II).

- *RMX-80 User's Guide*, 9800522, which describes how to operate the Intel Real-Time Mulitasking Executive (RMX-80).
- *RMX-80 Installation Guide*, 9803087-01, which describes installation and operation of the Intel Real-Time Multitasking Executive.

The following manuals may be required if you intend to call subroutines written in other Intel-supported languages:

- *8080/8085 Assembly Language Programming Manual,* 9800301, which describes the instructions and directives of the 8080/8085 assembler.
- *8080/8085 Macro Assembler Operator's Manual,* 9800292, which describes how to assemble (using ISIS-II) a program written in 8080/8085 assembly language.
- *PL/M-80 Programming Manual,* 9800268, which describes the instructions, conventions, and usage of PL/M-80, and how to create PL/M-80 programs.
- *ISIS-II PL/M-80 Compiler Operator's Manual,* 9800300, which describes how to use the ISIS-II based PL/M compiler to generate executable machine code.
- *FORTRAN-80 Programming Manual,* 9800481, which describes the instructions, conventions, and usage of FORTRAN-80, and how to create FORTRAN-80 programs.
- *ISIS-II FORTRAN-80 Compiler Operator's Manual,* 9800480, which describes how to use the ISIS-II based FORTRAN-80 compiler to generate executable machine code.

The following manuals are required to implement the RMX-80 sample configuration of BASIC-80 described in Appendix F:

- *iSBC 80/30 Single Board Computer Hardware Reference Manual,* 9800611, which describes how to configure the iSBC 80/30 under RMX-80.
- *iSBC 016 Random Access Memory Board Hardware Reference Manual,* 9800279, which describes how to map memory to fit the sample RMX-80 configuration.
- *iSBC 204 Flexible Diskette Controller Hardware Reference Manual,* 9800568, which describes how to configure the iSBC 204 for the sample RMX-80 configuration.
- *iSBC 80/10 Hardware Reference Manual,* 9800230.
- *iSBC 80/10A Hardware Reference Manual,* 9800230.
- *iSBC 80/20 Hardware Reference Manual,* 9800317.
- *iSBC 80/20-4 Hardware Reference Manual,* 9800317.

CONTENTS

ILLUSTRATIONS

TABLES

CHAPTER 1
INTRODUCTION TO BASIC-80

BASIC-80 is Intel's implementation of disk extended BASIC for Intellec and Intellec Series II Microcomputer Development Systems, which use the Intel Systems Implementation Supervisor (ISIS-II), and for Intel Single Board Computers, which use Intel's Real-Time Multitasking Executive (RMX/80). It offers a quick method of applying the computational and input/output capabilities of the microcomputer development system to a wide range of business, information handling, numeric analysis, and data processing applications.

BASIC-80 can be used with either the ISIS-II or RMX/80 operating systems. Requirements for ISIS-II BASIC-80 include an Intellec Microcomputer Development System with at least 48K of memory space, and one disk drive. RMX/80 BASIC-80 requirements, both hardware and software, are described in Appendix F of this manual.

BASIC-80 includes 21 commands, 39 statements, 48 functions, a line-editing capability, and full disk I/O (both sequential and random access). In addition, user-written functions can be defined, and up to 25 subroutines can be written in other Intel-supported languages (FORTRAN-80, PL/M-80, and 8080/8085 assembly language) and called from BASIC-80.

Intel integer, single-precision floating-point, and double-precision floating-point arithmetic standards are all supported, offering flexible combinations of processing speed and accuracy (up to 16 digits in the range $\pm 2.2 \times 10^{-308}$ to $\pm 1.8 \times 10^{308}$). Arrays can have virtually as many dimensions as needed; the only limit on the number of dimensions that can be specified is the 255-character program statement length.

Hexadecimal and octal integer representation, combined with bitwise Boolean logical operators, make sophisticated mask operations easy. A full range of string functions is available to provide flexibility in manipulating character data.

The disk I/O features include not only the ability to read from and write to disk files, but also the ability to create, rename, change the attributes of, delete, and list the directory of disk files without returning to the operating system.

BASIC-80 requires an Intellec or Intellec Series II microcomputer development system with at least 48K RAM and ISIS-II operating system (version 3.4 or later).

Operating System Interface

You can invoke BASIC-80 from ISIS-II, or configure RMX-80 BASIC-80 in PROM for boot loading upon restart. Once BASIC is running, you have access to many of the disk file-handling functions as well as the ability to load and save programs.

Invoking BASIC-80

Once you configure RMX/80 BASIC-80, you will always enter BASIC-80 upon restart.

To invoke BASIC-80 from ISIS-II, enter the name of the file that contains the BASIC interpreter. Options also allow you to specify the name of a file that contains a program to be loaded after BASIC-80 is running, and upper memory limit for BASIC-80's work area. The format of the command is:

```
-BASIC [filename] [MEMTOP(address)]
```

—

is the ISIS-II command prompt. It is displayed automatically before you enter the command.

BASIC

specifies the name of the file that contains the BASIC-80 interpreter.

filename

is an optional parameter that specifies the name of a file that is to be loaded and run after BASIC-80 is running.

MEMTOP(address)

is an optional parameter that specifies the upper bound of the memory that BASIC-80 can use. Address can be either a decimal or hexadecimal number. It must be greater than 3800H plus the number of bytes in the interpreter, and less than 0BEBFH in a 48K system or 0F6BFH in a 64K system.

Examples

1. If the interpreter is in a file named BASIC on a disk in drive 0 enter:

```
-BASIC
```

2. If the situation is the same as 1, but you want to run a file named ANLYZE on a disk in drive 1:

```
-BASIC  :F1:ANLYZE
```

3. If the situation is the same as 2, but you also want to prevent BASIC-80 from using memory beyond address 54400:

```
-BASIC  :F1:ANLYZE MEMTOP(54400)
```

If a fatal ISIS-II error occurs while BASIC-80 is running, ISIS-II is re-initialized and the contents of the BASIC-80 work area is lost, including any program editing you have done since you last entered a save command.

Manipulating Files from BASIC-80

BASIC-80 lets you list a disk directory, rename a file, change the attributes of a file, and delete a file. These functions can also be performed using ISIS-II or RMX-80, of course.

Listing the Directory of a Disk

To list the directory of a disk, enter DIR followed by the drive number. BASIC-80 assumes drive 0 if you don't specify:

```
DIR

NAME.EXT        RECS        LENGTH        ATTR
ALPHA            31          3728
ATTRIB           38          4677
BASIC           178         22571
COPY             64          7967
DCOPY            32          3961
DELETE           37          4501
DIR              46          5728
DSORT            11          1264
EDIT .MAC         5           469
INDEX .I         46          5669
NED              79           469
RENAME           21          2438
```

Renaming a File

The RENAME command lets you change the name of any file from an old filename to a new filename. The directory listing also changes to the new filename. In the example below, file :F1:PROG changes to :F1:MYPROG:

```
RENAME ":F1:PROG" to ":F1:MYPROG"
```

Changing File Attributes

With the BASIC-80 ATTRIB command, you can protect a file from overwriting, deletion, or being renamed by setting the write-protect attribute "W". If you set the invisibility attribute, "I", the file will not appear when the directory is listed. If you set the format attribute, "F", the file will copy to a disk formatted with the IDISK or FORMAT commands. The system attribute, "S", makes the specified file copy to a disk when the disk is formatted by the FORMAT command or copied with the COPY command.

For each attribute, the format is "X0" if the attribute is disabled, "X1" if it is enabled, where X is either W, I, S, or F. The format of ATTRIB is:

```
ATTRIB "filename", "X0"|"X1"
```

Deleting a File

If you want to get rid of a file or program, you can use the KILL command to delete it and remove its listing from the directory. Once a file or program has been killed, it cannot be recovered. The format of the KILL command is KILL followed by the filename in quotation marks. For example, to delete a file named ANLYZ on the disk in drive 1, enter:

```
KILL ":F1:ANLYZ"
```

Loading a Program

The LOAD command loads a BASIC-80 program from disk. The program can be stored in either ASCII or internal format. To load a program named ANLYZE from a disk in drive 1:

```
LOAD ":F1:ANLYZE"
OK
```

You can now run, list, or edit the program.

Saving a Program

The SAVE command copies your program from Intellec memory to disk. You must specify a filename, enclosed in quotation marks:

```
SAVE ":F1:ANLYZ"
```

The SAVE command can also be used to list the contents of the current file on a line printer or other output device. For example, to list the current file on a line printer, you would enter:

```
SAVE ":LP:",A
```

WARNING

You can only write data to *one* disk in any disk drive each time you invoke BASIC. If you write to a disk in a given drive, remove that disk and insert another, and try to write to the new disk, you lose all data on the new disk. The exception to this is on systems with more than one drive. It is permissible to change a disk on a drive if that drive has not been written to since another drive has been written to. As an example, if your BASIC-80 program writes on a file in drive 0, and you then remove that disk, insert another and write on it, the contents of that second disk will be lost. If, however, your program wrote on a disk in drive 1 between writing on the different disks in drive 0, there would be no problem.

There are no restrictions on reading from disks.

CHAPTER 2
LANGUAGE ELEMENTS

A BASIC-80 program consists of instructions, which tell BASIC-80 what to do, and data, which gives BASIC-80 the information necessary to do it. This chapter describes the different types of instructions and data, and shows how to represent them.

Instructions

BASIC-80 performs work by interpreting user-provided instructions. These instructions are divided into three categories: commands, statements, and functions. These instruction types are described in the following topics; the individual instructions are described in detail in Chapters 6 and 7.

Commands

Commands are executed as soon as you enter them; they alter or direct entire programs or files. Most commands can be used in program statements, but many of them halt program execution and force variables to zero or null.

Table 2-1 lists the BASIC-80 commands.

Statements

Statements are executed when they are encountered during program execution. They make up most of the instructions of a program. Most statements can be entered as commands.

Table 2-2 lists the BASIC-80 statements.

Table 2-1. BASIC-80 Commands

Command	Description	Example
ATTRIB	Changes the attributes of a file.	ATTRIB ":F1:STAT", "W1"
AUTO	Automatically numbers program statements.	AUTO 25,500
CLEAR	Sets aside memory for strings.	CLEAR 2000
CONT	Continues execution after BREAK.	CONT
DELETE	Deletes a line or lines from a program.	DELETE 700-875
DIR	Displays a list of all non-invisible files on a disk.	DIR 1
EDIT	Specifies a program statement to be changed.	EDIT 170
EXIT	Returns to operating system.	EXIT
KILL	Deletes a file from disk.	KILL ":F1:STAT"
LIST	Displays a line or lines of a program.	LIST 300-400
LOAD	Retrieves a file from disk.	LOAD ":F1:DATES"

Table 2-1. BASIC-80 Commands (Cont'd.)

Command	Description	Example
MERGE	Combines file program with current program.	MERGE ":F1:TIME"
NEW	Deletes current program, clears variables.	NEW
NULL	Specifies nulls added to a line.	NULL 20
PRUN	Executes program in ROM.	PRUN 4E00H
RENAME	Changes file name.	RENAME ":F1:SOUP" TO ":F1:NUTS"
RENUM	Changes program line numbers.	RENUM
RUN	Executes program.	RUN
SAVE	Stores program or file on disk.	SAVE ":F1:INVEN"
TRON	Turns on trace facility.	TRON
TROFF	Turns off trace facility.	TROFF
WIDTH	Changes width of display line.	WIDTH 80

Table 2-2. BASIC-80 Statements

Statement	Description	Example
CLOSE	Closes one or more files.	CLOSE 3
DATA	Identifies values that can be assigned with a READ statement.	DATA 9, 0, "JUNE", .33
DEF	Defines a user-written function.	DEF FNRT (R1, R2) = R1*R2/(R1 + R2)
DEFDBL	Defines variable names starting with the given letter as double-precision floating-point.	DEFDBL R-Z
DEFINT	Defines variable names starting with the given letter as integer.	DEFINT I-N
DEFSNG	Defines variable names starting with the given letter as single-precision floating point.	DEFSNG B-H, X, Y
DEFSTR	Defines variable names starting with the given letter as string variable names.	DEFSTR K-O
DEFUSR	Defines non-BASIC subroutine.	DEFUSR 0 = 4E00H
DIM	Allocates space for array variables.	DIM char (25, 10, 25)
END	Concludes program.	END
ERROR	Simulates errors with given error number.	ERROR 12
FIELD	Allocates space in random file buffer.	FIELD #3, 20 AS A$
FOR-NEXT-STEP	Creates a loop.	FOR I = 1 TO 5 STEP.5 NEXT I
GET	Retrieves data from disk file.	GET #2, 4
GOSUB	Transfers execution to subroutine.	GOSUB 550
GOTO	Transfers execution to line number.	GOTO 400
IF-THEN-ELSE	When the expression specified is true, the statement executes; if false, a second statement executes.	IF A>B THEN 250 ELSE PRINT ">"
INPUT	1. Prompts for terminal input in program	INPUT A, B, C
	2. Reads data from sequential file.	INPUT #1, A$, B$, C$
LET	Assigns value to variables.	LET A =52

Table 2-2. BASIC-80 Statements (Cont'd.)

Statement	Description	Example
LINE INPUT	Enters entire line from a disk file.	LINE INPUT A$
LSET	Left justifies text in random file buffer.	LSET A$ = B$
ON ERROR	Traps errors by branching to error-resolving routines.	ON ERROR GOTO 900
ON-GOTO	Transfers execution to Xth line number for expression X.	ON X GOTO 460, 480
ON-GOSUB	Transfers execution to Xth subroutine for expression X.	ON X GOSUB 220, 240, 260
OPEN	Creates sequential or random disk files.	OPEN "R", 1, ":F1:TRACE"
OPTION BASE	Starts arrays at 0 or 1.	OPTION BASE 0
OUT	Writes values to I/O ports.	OUT 00F0, 12
POKE	Writes byte to memory location.	POKE 0A077, 72
PRINT	1. Displays text on terminal. 2. Stores data in sequential disk file.	PRINT A, B, C PRINT #4, A$, B$, C$
PRINT USING	Displays text according to given format.	PRINT USING "$$##,##; 125.38, 21.14. 6.10
PUT	Stores data in random disk file.	PUT #3, A$, B$ C$
RANDOMIZE	Initializes random number generator.	RANDOMIZE
READ	Assigns values from DATA statements to program variables.	READ A, K1, L%, Z
REM	Comments in program text	10 REM THIS IS 20 REM A REMARK
RESTORE	Resets pointer for reading DATA statements.	RESTORE
RESUME	Restarts execution after errors.	RESUME
RETURN	Transfers control back to statement following last GOSUB.	RETURN
RSET	Right justifies text in random file buffer.	RSET L$ = MK$
STOP	Halts program execution.	STOP
SWAP	Exchanges values of two variables of similar type.	SWAP A1#, B2#
WAIT	Halts execution until port changes.	WAIT 1, 04H, 0AH

Table 2-3. BASIC-80 Functions

Functions	Returns	Example
ABS	Absolute value.	ABS (X)
ASC	ASCII code of the first character of the specified string.	ASC (A$)
ATN	Arctangent, in radians.	ATN (X)
CDBL	Double-precision floating-point value.	CDBL (X)
CHR$	Character corresponding to the specified ASCII code.	CHR$ (X)
CINT	Integer value	CINT (X)
COS	Cosine, in radians	COS (X)
CSNG	Single-precision floating-point value.	CSNG (X)
CVD	Double-precision floating-point value equal to 8-byte string A$	CVD (X#)
CVI	Integer value equal to 2-byte string A$	CVI (X%)
CVS	Single-precision floating-point value equal to 4-byte string A$	CVS (X!)

Table 2-3. BASIC-80 Functions (Cont'd.)

Functions	Returns	Example
DSKF	Number of 128-byte sectors free on disk or drive (X)	DSKF (X)
EOF	–1 if end-of-file; 0 if not (for file X).	A = EOF (X)
ERL	Line number of last error.	ERL
ERR	Error code of last error.	ERR
EXP	e to the (X)th power.	EXP (X)
FIX	Integer value of (X).	FIX (X)
FRE	Number of bytes in memory (X) or number of bytes in string space (X$)	FRE (X)
HEX$	String equal to hex value of (X%)	HEX$ (X%)
INP	Reads a byte from port (X)	INP (X)
INPUT$	Inputs (X) characters from file (Y)	INPUT$ (X, Y)
INSTR$	Position of (X$) within (Y$)	INSTR$ (X$, Y$)
INT	Integer value of (X)	INT (X)
LEFT$	Leftmost (X) characters of (A$)	LEFT$ (A$, X)
LEN	Character length of (X$)	LEN (X$)
LOC	Current record number in random file X. Sectors read or written since last OPEN in sequential file X.	LOC (X)
LOF	Number of records in random file X. Number of data sectors in sequential file X.	LOF (X)
LOG	Natural log of (X)	LOG (X)
MID$	J characters, starting at I, of string A$	MID$ (A$, I, J)
MKD$	8-byte string equal to double-precision floating-point variable (X!)	MKD$ (A$)
MKI$	2-byte string equal to integer variable (X%)	MKI$ (A$)
MKS$	4-byte string equal to single-precision floating-point variable (X!)	MKS$ (A$)
OCT$	Octal equivalent of decimal argument	OCT$ (X)
PEEK	Single byte from memory location (X)	PEEK (X)
POS	Position of cursor after last PRINT.	POS (dummy argument)
RIGHT$	Rightmost (I%) characters of X$	RIGHT$ (X$, I%)
RND	Single-precision random number between 0 and 1.	RND
SGN	Sign of (X)	SGN (X)
SIN	Sine of (X)	SIN (X)
SPACE$	String of (I%) spaces	SPACE$ (I%)
SPC	String of (X%) spaces	SPC (X%)
SQR	Square root of (X)	SQR (X)
STR$	String equal to (X)	STR$ (X)
STRING$	Character X, Y% times—or the first character of A$, Y% times.	STRING$ (Y%, X) STRING$ (Y%, A$)
TAB	Spaces to (X) position on terminal	TAB (X)
TAN	Tangent value of (X)	TAN (X)
USR	References user subroutine 0 to 24	AX = USR 12 (A1, A2)
VAL	Numerical value of (X$)	VAL (X$)
VARPTR	Memory address of (X)	VARPTR (X)

Functions

Functions are built-in routines that return a value based on the argument or arguments supplied. They can be used to form expressions with either commands or statements. BASIC-80 includes both numeric and string functions.

Table 2-3 lists the BASIC-80 functions. In addition to these, up to 10 user-written functions can be defined with the DEFFN statement.

Representing Data

The instructions described in the previous topics tell BASIC-80 *what* to do; to carry out these instructions, you must also provide data in a specific fashion. Intel BASIC-80 includes constant and variable values, in either numeric or string format; allows these values to be grouped into arrays; provides for conversion from one data type to another; and allows these values to be combined into expressions using arithmetic, relational, and logical operators.

Syntax

BASIC-80 accepts instructions and data in a specific format. This format, called syntax, must be followed to obtain useful, predictable results. BASIC-80 syntax is a superset of ANSI Minimal BASIC syntax. The table below describes the meta-language elements used to illustrate BASIC-80 syntax.

Table 2-4. BASIC-80 Metalanguage Elements

Condition	Example
An instruction that requires no argument is shown by itself in uppercase letters.	RESTORE
If an argument must be provided, the description of the argument, in lowercase letters, follows the instruction.	GOTO line POKE address, value
If an argument is optional, the description of the argument is enclosed in brackets.	RESUME [line number] SAVE "filename" [,A]
If more than one type of argument can be specified, the choices are separated by vertical lines.	PRINT expression|variable
If an argument can be repeated, three dots signify repetition.	READ data [,data]... ON variable GOSUB line [,line]...

Numeric Data

BASIC-80 accepts numeric values as either constants or variables. Within these two categories, there are three types of representation: integer, single-precision floating-point, and double-precision floating-point.

Using the DEFINT, DEFSNG, or DEFDBL statements, you can define a range of letters to signify integer, single-precision floating-point, or double-precision floating-point numeric variables. If you don't define numeric type, you can specify it with a one-character suffix when you use the variable or constant letter name. If you don't specify numeric type, the default is single-precision floating-point, as if a DEFSNG A-Z instruction had been given.

Table 2-5 summarizes the characteristics and methods of specifying numeric data types.

Constants

Constants are numeric values that do not change during program execution. A constant can be a decimal integer, hexadecimal integer, octal integer, single-precision floating-point number, or double-precision floating-point number.

Table 2-5. Numeric Data Types

Numeric Type	Range	Storage Required	Definition	Suffix	Examples
Integer (decimal)	−32768 to +32767	2 bytes	DEFINT	%	X% 9463%
Integer (hexadecimal)	0 to FFFFH	2 bytes	—	H	0FF4H
Integer (octal)	0 to 177777Q	2 bytes	—	Q	772Q
Single-precision floating-point (7 digits precision)	$\pm 1.2 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$	4 bytes	DEFSNG	!	X! 9436.5! 9.4365E03
Double-precision floating-point (16 digits precision)	$\pm 2.2 \times 10^{-308}$ to $\pm 1.8 \times 10^{308}$	8 bytes	DEFDBL	#	X# 9436.5# 9.4365D03

Integer Constants

Integer constants are whole numbers in the range −32768 to 32767. Each integer constant requires two bytes of memory. Because the storage requirements are lowest for integers and integer arithmetic is much faster than floating-point arithmetic, it's a good idea to use integer representation wherever possible.

Decimal Integer Constants

To identify a constant as a decimal integer constant, add the suffix % to the decimal integer value.

Some decimal integer constants are:

```
23%
-373%
```

Hexadecimal Integer Constants

Hexadecimal integer constants are identified by the suffix H following the numeric value. The characters 0-9 and A-F (representing the decimal values 10-15) are used as hexadecimal digits. Each character represents 4 bits of data. The first character must be a decimal digit, so it may be necessary to add a leading 0. Some hexadecimal integer constants are:

Hexadecimal	Decimal Equivalent
1FH	31
0C76H	3190
7H	7

Octal Integer Constants

Octal integer constants are identified by the suffix Q following the numeric value. The numerals 0-7 are used as octal digits. Each digit represents 3 bits of data. Some octal integer constants are:

Octal	Decimal Equivalent
772Q	506
4444Q	2340
7Q	7

Single-Precision Floating-Point Constants

Single-precision floating-point constants are identified by the suffix ! following the numeric value, by the letter E identifying the exponent in scientific notation, or by the presence of a decimal point in a number having seven or fewer digits. Floating-point numbers in the range $\pm 1.2 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$ are represented with seven digits of accuracy.

Each single-precision floating-point constant requires four bytes of memory. Because this is half the storage required by double-precision floating-point constants, and because single-precision arithmetic is quicker than double-precision arithmetic, it's a good idea to use single-precision wherever possible for floating-point operators.

Some single-precision floating-point constants are:

```
142!
-1.414
6.259371E-09
```

Double-Precision Floating-Point Constants

Double-precision floating-point constants are identified by the suffix # following the numeric value, by the letter D identifying the exponent in scientific notation, or by having more than seven digits. Floating-point numbers in the range $\pm 2.2 \times 10^{-308}$ to $\pm 1.8 \times 10^{308}$ are represented with 16 digits of accuracy.

Each double-precision floating-point constant requires eight bytes of memory. Some double-precision floating-point constants are:

```
-2.001317921D12
11235813213455
24.2#
```

Variables

Numeric variables represent numeric values that can change during program execution. These can be of three types, like numeric constants: integer, single-precision floating-point, or double-precision floating-point. Numeric variables are represented by one or two characters followed by an optional type identifier suffix. The first character must be a letter; the second, which is optional, may be any alphanumeric character. If the variable name contains more than two characters besides a type identifier suffix, the rest of the variable name characters are ignored. No words used as BASIC-80 instruction words may be used within variable names.

If a variable is referenced before it has been assigned a value, its value is zero. The NEW, RUN, CLEAR, LOAD, and MERGE instructions set all variables to zero.

Individual variables can be specified by individual type identifier suffixes, which override group type identifiers used to specify blocks of variables. Table 2-5 shows these suffixes.

Blocks of variables beginning with specific characters can be specified as integer, single-precision, or double-precision with the DEFINT, DEFSNG, and DEFDBL statements. The general form of these statements is: DEFxxx m[-n], where n is any letter A through Z, and m is any letter A through Z that precedes n in the alphabet, (i.e., the block L-Q is legal, but Q-L is not). In this way, all variables beginning with a certain letter or letters may be defined as one type.

The variable default type is single precision, as if a DEFSNG A-Z had been executed at the start of a given program. If certain variables should be of another type, you should define them at the start of the program to prevent errors. In all cases, the type identifiers (% for integer, ! for single-precision, and # for double-precision) override any variable block type assignment.

Note that A$, A%, A!, and A# are four different variables. If the default variable type for variables beginning with the letter A is single precision, then A and A! are the same variable.

To economize on memory space and execution time, you should use integer representation rather than single-precision representation, and single-precision rather than double-precision, when this is possible.

String Data

BASIC-80 accepts strings of characters as data. Like numeric values, strings can be either constants or variables.

String Constants

A string constant is a group of characters, enclosed in quotation marks. Quotation marks cannot be used within string constants. String constants can be up to 255 characters long. Some string constants are:

```
"This is a string constant."
"48, 23H, 373799"
```

String Variables

String variables are string values which can change during program execution. A string variable name is one or more characters, the first of which must be a letter, followed by $. If more than two characters are entered as a variable name, onlv the first two are read.

String variables can contain strings of from 0-255 characters. When you first invoke BASIC-80, however, there is only storage space for 100 characters. The CLEAR command must be used to increase the amount of available string space. Here are some examples of assignments to string variables:

```
A$ = "Enter next data string"
B$ = "40 * 1.7234E+3"
NAMES$ = "Warren, Mark, Evan"
```

Converting Data

It is sometimes useful to convert one type of data into another. BASIC-80 supports these conversions with the HEX$, OCT$, CHR$, STR$, VAL, CVD, CVI, CVS, CDBL, CSNG, CINT, MKS$, MKD$, and MKI$ functions.

The HEX$ and OCT$ functions return a string of hexadecimal and octal digits, respectively, that represent the numeric argument. The STR$ function returns a string of decimal digits that represent the decimal value of the argument. The VAL function returns the numeric value of the string argument, if the string is a number.

CHR$ returns the ASCII equivalent of an integer argument, between 0 and 255.

The CVI, CVS, and CVD functions convert a given string into integer, single- precision floating-point, or double-precision floating-point numeric values, respectively. These functions are used to retrieve numeric values from the input/output buffer when doing random disk I/O.

The MKI$, MKS$, and MKD$ functions convert integer, single-precision floating-point, and double-precision floating-point numeric values, respectively, into a string. These functions are used to store numeric values in the input/output buffer when doing random disk I/O.

You can convert a numeric variable (integer, single-precision floating-point, or double-precision floating-point) to any of these types by using the variable in an expression with the CINT, CDBL, or CSNG functions:

```
A# = CDBL (A%)
L4! = CSNG (L4)
VAR5 = CINT (VAR5)
```

Array Variables

An array is a group of variables identified by the same name, specified by subscripts that define their position in the array. An array variable can have as many dimensions as will fit on a single line. An array variable is specified by following a variable name with as many subscripts as there are dimensions. A subscript must be an integer value, and enclosed within parentheses or square brackets. If there is more than one subscript, separate them with a comma. Expressions can be used to specify subscripts; they are rounded to integer form. Here are some array variables:

```
X (10)
R1(5,4)
Y(I,1)
BA(I + 3,X(10))
```

BASIC-80 normally indexes arrays from zero; that is; the first element in an array is defined as 0. To start arrays at one in BASIC-80, enter the instruction OPTION BASE 1 in your program before you dimension or reference any arrays.

The DIM statement allocates array space and specifies the maximum allowable subscript for a given dimension. If an array variable is referenced before it has been formally dimensioned, BASIC-80 allocates an index of 10 for each dimension. Some examples of the DIM statement:

```
DIM X(15)
DIM R1(12,8)
DIM K(17,24)
```

An attempt to specify an array variable whose subscripts are larger than the dimensioned value, or which exceed 10 in the default mode, causes a SUBSCRIPT OUT OF RANGE error message.

String Arrays

Like numeric arrays, string arrays can be dimensioned with the DIM statement. The format for dimensioning a string array is the same as for numeric arrays:

```
DIM A$(5,25,40)
```

If you don't execute a DIM statement, a default of 10 for each subscript is assumed. If this value is then exceeded, an error message will result.

Table 2-6. BASIC-80 Operators in Order of Precedence

Order	Operator	Example
1.	Expressions in parentheses.	(A+B)
2.	Exponentiation, as shown in the example, where A is raised to the B power.	A↑B
3.	Negation, represented by the minus sign.	−A
4.	Multiplication and Division, represented by an asterisk (*) and a slash (/) respectively.	A*B A/B
5.	Integer division, represented by a backslash ( ). Both arguments are converted to integer values and the result is truncated to an integer.	A \ B
6.	Integer Modulus, represented by MOD. Both arguments are converted into integers. The result is the remainder when the first is divided by the second.	A MOD B
7.	Addition and Subtraction, represented by (+) and minus (−) signs.	A+B A−B
8.	Relational Operators. These are listed without precedence. For all relational operators, the result is −1 if true, and 0 if false. The arguments A and B must be both strings or both numeric variables.	
	Equals sign: Used to test for equality.	A=B
	Greater Than: Used to test magnitude between two arguments. The large end of the sign faces the posited greater value.	A>B
	Less Than: Used to test magnitude between two arguments. The small end of the sign faces the posited lesser value.	A<B
	Not Equal: Used to test for inequality between two arguments.	A><B A<>B
	Greater Than or Equal To: Used to test magnitude down to the level of the second argument.	A=>B A>=B
	Less Than or Equal To: Used to test magnitude up to the value of the second argument.	A=<B A<=B
In the Logical Operators below, the arguments are converted to 16-bit, signed two's complement integers in the range −32768 to +32767. The result of the operation is converted to the same format. The operations are performed one bit at a time, comparing the nth bit of X with the nth bit of Y.		

Table 2-6. BASIC-80 Operators in Order of Precedence (Cont'd.)

Order	Operator	Example
9.	Logical NOT, used to invert a given argument.	NOT −1 = 0
10.	Logical AND, used to test if the nth bit of X and the nth bit of Y are both on.	1 AND 0 = 0
11.	Logical OR, used to test if the nth bit of X or Y equals 1.	15 OR 0 = 15
12.	Logical exclusive OR, used to test if either the nth bit of X or the nth bit of Y = 1, but not both.	15 XOR 7 = 8
13.	Logical implication, used to test if the nth bit of X is on, then the nth bit of Y is on.	0F0F0H IMP 00FFH=0FFF0H
14.	Logical equivalence, used to test if the nth bit of X equals the nth bit of Y.	0F0FH EQV 00FFH=0F00FH

Operators and Precedence of Evaluation

Complex expressions may be formed by combining constants and variables with arithmetic, logical, relational, and string operators. BASIC-80 follows an order of precedence to insure orderly and predictable evaluation when analyzing complex expressions. This order of precedence may be overridden by parentheses; any elements within the parentheses are evaluated first. The numeric operators (arithmetic, logical, and relational) are listed in order of precedence in Table 2-6.

Arithmetic Operators

There are seven arithmetic operators in BASIC-80, each performing a familiar arithmetic operation on two numeric expressions. They are evaluated before the relational or logical operators, and if two operators of equal precedence are found by BASIC-80, they are evaluated from left to right. Table 2-6 lists the arithmetic operators in order of precedence. Some examples of arithmetic operators are:

```
A  = B*(C*2.49)
K1 = (L+M)  5
R5 = (B3*E↑.5)
```

Relational Operators

There are six relational operators in BASIC-80, which test relationships between two expressions and return a −1 if the premise is true, a 0 if it is false. You can write instructions to direct program execution according to either result. The relational operators are evaluated after the arithmetic operators, and if two operators with the same order of precedence are given in an expression, they are evaluated left to right. Table 2-6 lists the relational operators, and some examples are given below:

```
IF (A*2.2)<>B*B1 THEN 220
IF INT(A1) = INT(B1) THEN A=B
IF A> B THEN IF B>(C* VA#)THEN 340
```

Logical Operators

The logical operators NOT, AND, OR, XOR, IMP, and EQV are operators that compare the nth bit of argument X with the nth bit of argument Y. They are evaluated after the arithmetic and relational operators; therefore, arithmetic expressions resolve to a number which is compared with another number. A relational operator test can be used with logical operations. If there are two logical operators of the same precedence in a single expression, they are evaluated left to right. Table 2-6 lists the logical operators in order of precedence. Some examples of the logical operators, used in complex expressions, are shown below.

```
IF A>=3.5 X OR X > 3 THEN Q1=0
IF B=1 OR B=2 OR B=3 THEN 2750 ELSE 280u
IF (A-B) AND (B + 3) THEN STOP ELSE IF (A AND B) THEN CLOSE 2
```

String Operator

The relational operators may be used with strings to compare them according to ASCII order. If strings of unequal length are compared, and the shorter is identical to the first part of the longer, then the longer is greater. There is one operator only used with strings: the concatenation operator (+). This operator defines a string as two or more strings joined together.

```
A$ = B$ + C$
```

Expressions

Except for the command and statement instructions, all of the language elements previously discussed can be combined to form expressions. Depending on the type of constants, variables, and operators used, expressions can be classed as numeric or string.

Numeric Expressions

In BASIC-80, numeric expressions are created with numeric variables, constants, functions, and operators. Variables are initialized with 0, and may be assigned other values with assignment statements, or with INPUT statements during program execution.

Any function which returns a numeric value can be used in a numeric expression. Strings can only be used if they are converted to a number. Numeric expressions can use arithmetic, logical, or relational operators. Some numeric expressions:

```
K(I) = B*SQR(X)
IF A>12.1 THEN C = C + 1
IF PEEK (2FFFH) AND 0CH THEN PRINT "ON"
```

String Expressions

String expressions can be specified in BASIC-80 using string constants, string variables, relational operators, and the concatenation operator (+). The concatenation operator combines two strings into one. If the resulting string is longer than 255 characters, execution halts and an error message is displayed. Some string expressions are:

```
A$ = "NAME:" + NAME$
IF B1$>R$ THEN B1$ = ""
R$(I)=R$(I) + S$(I) + "DONE"
```

CHAPTER 3
ENTERING AND EDITING PROGRAMS

With BASIC-80, you can create new programs by entering statements line by line, or you can access saved programs from disk storage. If you're using ISIS-II BASIC-80, you can use the ISIS-II BASIC-80 Text Editor to alter new or saved instruction lines. RMX/80 BASIC-80 does not have a Text Editor.

The following topics show how to use BASIC-80 programming features and the ISIS-II BASIC-80 Text Editor to aid program development.

Entering Instruction Lines

When you invoke BASIC without specifying a file name, there is no program to run. The system is ready to accept commands or program statements. A statement consists of a line number from 0 to 65529 followed by the language elements (program statement, constants, variables, operators, functions, etc.). If you type a line number alone after a line with that number has been entered with text, that line is deleted.

You can enter statements in any order. To review the statements in a program, use the LIST command. It displays the statement in numeric order.

You can have BASIC-80 provide line numbers, starting at a given number, with a given increment, by using the AUTO command. After you enter the AUTO command, BASIC displays the line number and waits for you to enter the statement. When you end the statement with a carriage return (CR), it prints the next line number and again waits. To stop the automatic line numbers, enter a Control-C.

If AUTO generates a line number that already exists in the program, it prints an asterisk (*) after the line number. If you enter a statement, what you enter replaces the existing statement. If you enter a Control-C, the existing line is unchanged.

You can use Control-I as a tab key if you want to format your statements. The width of each line is divided into 8-character-wide columns. Each time you press Control-I, the cursor or print head moves to the beginning of the next column.

BASIC assumes a width of 72 characters. You can change the width with the WIDTH command.

To enter more than one statement per line, separate each statement with a colon (:). If you want to format the program so that additional statements appear on separate lines (but are still part of the same numbered program statement), use the Line Feed (LF) key to move to the beginning of the next display line. You can do this any number of times, up to the 255-character line-length limit; the program statement doesn't end until you enter CR.

To put a comment in a BASIC-80 program, enter REM after the line number. BASIC-80 doesn't try to execute such lines, but they become part of the program.

Correcting Entry Errors

If you make an error while entering a line, you can correct it by using the RUBOUT key to erase characters (as long as you haven't entered the line into memory by pressing CR).

In the Command Mode, the RUBOUT key deletes the last entered character each time you press it, and backspaces the cursor on a CRT. On a teletype, or with RMX/80 BASIC-80, RUBOUT echoes the last-entered character. If you then press CR, the program statement is entered without the rubbed-out characters. If you enter new characters and then press CR, the new characters appear in the line.

Suppose you enter 52 instead of 55.To erase the 2, press RUBOUT:

```
30 A=B*52 ■
30 A=B*5 ■
```

If you press RUBOUT again, the 5 is deleted:

```
30 A=B* ■
```

To change 52 to 37, press RUBOUT twice, then 3, 7:

```
30 A=B*52 ■
30 A=B* ■
30 A=B*37 ■
```

When using the Edit Mode, RUBOUT works somewhat differently. Refer to the Editing Program Text Section for details.

Control-R displays the line as corrected, still waiting for more input:

```
30 A--=Bxx*522537 (Control-R)
   A = B*37
```

To cancel a line, simply press Control-X.

Editing Program Text

Intel ISIS-II BASIC-80 has an Editing Mode used to change individual characters, or entire lines. The Editing Mode has its own set of subcommands and syntax. Table 3-1 briefly describes each of the subcommands. If an illegal character is entered, the terminal beeps and the character is ignored.

Table 3-1. BASIC-80 Editing Subcommands

Function	Syntax
To delete text:	[integer] D
To insert text:	I character [character...]
To delete all characters to the right of the cursor and insert text:	H character [character...]
To insert characters at the end of the line:	X character [character...]
To search for a character:	[integer] S character
To delete all characters until specified character:	[integer] K character
To change next n characters to y characters:	[integer] C character [character...]
To restore original line and leave Edit Mode:	Q
To restore original line and restart Edit Mode:	A
To print balance of line and restart Edit Mode:	L
To leave Edit Mode and keep changes:	E
To leave Edit Mode, keep changes, and print the edited line:	Carriage return
To delete unwanted characters:	Rubout
To place one logical statement line on two or more physical lines:	Line Feed
To leave next n characters unchanged:	[integer] space

Complete editing of a line replaces the old line with the edited line. This resets all variables to zero or null. To end editing without losing prior variable values, exit the editing mode with the Q subcommand after the line number has been printed. BASIC-80 returns to command level, variable values are unchanged and any editing changes are lost.

There are three ways to enter the Editing Mode:

1) Type EDIT line number, and BASIC-80 returns the line number requested. If you press the space bar, the cursor moves to the desired location in the instruction line.
2) When entering text, type a Control A instead of a carriage return. This causes a carriage return, a space, and the computer prints an exclamation point. The cursor points at the first character of the sequence, and can be advanced by pressing the space bar. If you use CONTROL-A after listing a program, it edits the last line, and you can't change line numbers.
3) If BASIC-80 encounters an instruction line with a syntax error during program execution, it will halt and print an error message of the format: SYNTAX ERROR IN (line number). Below the error message, the computer returns the line number, and the line may be edited.

   In the Edit Mode, pressing RUBOUT will echo characters, but they are not deleted. Use the D subcommand to delete characters in the Edit Mode.

In the explanatory sections below, a typical line of program text is edited and re-edited by each subcommand. The "■" indicates the position of the cursor or print head, and all characters are shown as they appear on the terminal. Such editing subcommands as D, L, Carriage Return, Escape, and so on are represented in parentheses: (CR), (ESC), (D), (4SE) to avoid confusion. You should try out these or other examples to gain facility with the editor.

In the following sequence of edit subcommands, we will be editing line 40 of a hypothetical program. Line 40 returns a syntax error message, since it needs a PRINT instruction following ELSE, and should not contain OR:

```
40 IF A>B THEN 120 OR ELSE "NULL SET"
```

D Subcommand

The D subcommand is used to delete characters to the right of the cursor. Spaces are counted as characters. If there are less than n characters to the right of the cursor, just the remaining characters are deleted. The argument n is an integer in the range 1 to 255, and the default value is 1. The deleted characters are printed enclosed by backslashes, i.e., \characters\ .

The syntax of the D subcommand is:

```
[integer] D
```

In the example below, line 40 returned a SYNTAX ERROR message when the program ran. BASIC-80 displays the error message and enters the Edit Mode. By pressing the space bar, the text of line 40 is displayed character by character until the incorrect character is encountered.

```
40 IF A>B THEN 120 OR ELSE "NULL SET"
40 IF A>B THEN 120■
```

The command(3D) (press 3, then D) results in:

```
40 IF A>B THEN 120 \OR\ ■
```

L Subcommand

The L subcommand prints the rest of the original line, and waits for further editing subcommands. The cursor is to the left of the first character. You can use the L subcommand to display previously edited text and restart editing at the beginning of the line:

```
40 IF A>B THEN 120 ELSE "NULL SET"
40 ■
```

I Subcommand

The I subcommand inserts characters after the last character typed. Each character typed after typing I is inserted at the current cursor position. To end insertion, press the ESCAPE key. To end insertion and leave the Editing Mode, press the Carriage Return key. Characters may be deleted when using the I subcommand by pressing the RUBOUT key.

The syntax of the I subcommand is:

```
I character [character]...
```

Suppose you want to insert the word "PRINT" into the previous example. Press the space bar until you reach the proper point:

```
40 IF A>B THEN 120 ELSE ■
and then enter:
(I) PRINT (ESC) (L)
And you will see:
40 IF A>B THEN 120 ELSE PRINT "NULL SET"
```

H Subcommand

The H Subcommand deletes all characters to the right of the cursor, and then enters the insertion mode, like the I Subcommand. When through inserting characters, enter (ESC) to end insertion or (CR) to end editing.

The syntax of the H subcommand is:

```
H Character[Character]...
```

If you want to change the message "NULL SET" in the previous example to "UNDEFINED SET", you can use the H Subcommand to do it. Move the cursor to the proper point with the space bar:

```
40 IF A>B THEN 120 ELSE PRINT

  Enter (H) "UNDEFINED SET" (ESC) (L):

40 IF A>B THEN 120 ELSE PRINT "UNDEFINED SET"
```

X Subcommand

The X subcommand prints the rest of the line to be edited, moves the cursor to the end of the line, and enters the insertion mode, as for the I subcommand. This subcommand is used to add new text at the end of instruction lines. Execution is as in the insertion mode.

The syntax of the X subcommand is:

```
X character [character]...
```

Returning to the previous example, if you wish to add text at the end of the given instruction line, use the X subcommand:

```
40 ■
  Enter (X)

40 IF A>B THEN 120 ELSE PRINT "UNDEFINED SET" ■

  Enter new text at the cursor—;A; B (ESC) (L):

40 IF A >B THEN 120 ELSE PRINT "UNDEFINED SET";A; B
```

S Subcommand

The S subcommand examines characters to the right of the cursor to find the nth occurrence of the specified character,where n is an integer in the range 1 to 255; the default is 1. This subcommand skips the first character to the right of the cursor and searches, printing all characters encountered. When the nth occurrence of the specified character is found, the cursor stops at the character. If the nth occurrence of the specified character is not found, the cursor stops at the end of the line.

The syntax of the S subcommand is:

```
[integer] S character
```

The S subcommand can be used with an example to find the nth occurrence of the specified character. Suppose we want to move the cursor to the space occupied by the 3rd letter "E" in line 40:

```
40 ■
  Enter (3SE):

40 IF A>B THEN 120 ELSE ■
```

At this point, the other editing subcommands may be used.

K Subcommand

The K subcommand functions like the S subcommand except that it deletes all characters passed over until the nth occurrence of the specified character. The deleted characters are enclosed in backslashes.

The syntax of the K subcommand is:

```
[integer] K character
```

The K subcommand may be used on our example. It will eliminate all text up to the 1st occurrence of P, and print backslashes:

```
40 ■
  Enter (1KP)

\40\ IF A>B THEN 120 ELSE ■
  Enter (L)

40 PRINT "UNDEFINED SET";A; B
```

C Subcommand

The C subcommand changes the next n characters to the specified character(s). If no integer is specified, the character immediately to the right of the cursor is changed.

The syntax of the C subcommand is:

```
[integer] C character [character...]
```

In our previous example, line 40 was reduced to:

```
40 PRINT "UNDEFINED SET";A; B
```

This can be changed to print "REDEFINED SET" with C:

```
Move the cursor to:
40 PRINT ■

  Enter (2C) RE (L)

40 PRINT "REDEFINED SET"; A;B

40 ■
```

Q Subcommand

The Q subcommand restores the original line, as it was prior to editing, and leaves the editing mode. Note that if an instruction is edited with Q, the changes are lost. This subcommand returns the user to the command mode without executing an edited command. This subcommand only works when editing program lines.

A Subcommand

The A subcommand restores the original line and prints the original line number below. Editing is restarted. This subcommand only works when editing program lines.

When the A subcommand is used with the previous example, the result is:

```
40 PRINT "UNDEFINED SET" ■
Enter (A) (L)
40 IF A>B THEN 120 ELSE "NULL SET"
40■
```

E Subcommand

The E subcommand exits the editing mode. The edited line replaces the original line.

When the E subcommand is used with the previous example, the result is:

```
40 PRINT "UNDEFINED SET"
 (The E subcommand is entered)
```

Carriage Return

A Carriage Return exits the editing mode, and prints the rest of the line being edited. The edited line replaces the original line.

CHAPTER 4
ERROR HANDLING

If you enter improper instructions, syntax, or formats, BASIC-80 issues an error message. This chapter explains what these errors mean, how they may be trapped with the ON ERROR statement and pinpointed with the TRON, TROFF, ERR, and ERL instructions, and how errors may be simulated with the ERROR statement.

BASIC-80 Error Messages

When BASIC-80 encounters an error in a program, it displays an error message. Appendix A lists the error messages and error codes, and describes their meaning. These errors stop program execution. You can use the BASIC-80 editor at this point to correct obvious errors, add error-trapping instructions, or use the trace facility.

Syntax Error Messages

If BASIC-80 detects a syntax error, it stops program execution, displays an error message, and enters the Edit Mode:

```
10 LET A=
20 PRINT A
RUN
SYNTAX ERROR IN 10
10 ■
```

You can now change line 10, as described in Chapter 3. To leave the Edit Mode without making any changes, and to preserve the variable values in the program, type Q. If you enter any other editing subcommands (including Carriage Return) variable values are lost.

Overflow, Underflow, and Divide-by-Zero

In BASIC-80, the single- and double-precision floating-point overflow, underflow, and divide-by-zero errors do not halt program execution.

Overflow. Single-precision floating-point overflow occurs when the magnitude of a single-precision numeric value exceeds (±) 3.4×10^{38} (3.4E38). Double-precision floating-point overflow occurs when the magnitude of a double-precision numeric value exceeds (±) 1.79×10^{308} (1.79D308). When a value of this magnitude is generated, the message OVERFLOW is displayed, a value equal to the largest possible magnitude for the given representation with the appropriate sign is assigned, and execution proceeds.

The following examples show single- and double-precision overflow:

```
10 A=1E20*1E20
20 D#=1D200*1D200
30 PRINT A;D#
RUN
OVERFLOW
OVERFLOW
3.402824E+38   1.79769313486231D+308
```

Underflow. Single-precision floating-point underflow occurs when the magnitude of a single-precision numeric value is less than (±) 1.2×10^{-38} (1.2E-38). Double-precision floating-point underflow occurs when the magnitude of a double-precision numeric value is less than (±) 2.2×10^{-308} (2.2D-308). When a value of this magnitude is generated, no message is printed, the result is defined as zero, and program execution continues.

Divide-By-Zero. Divide-by-zero is handled identically to single- and double-precision overflow protocol, except that the message printed is DIVIDE-BY-ZERO.

Integer Operations

Integer overflow and division by zero are treated as normal errors. Program execution halts. Integer overflow occurs when a value greater than 32767.49 or less than −32768.49 is converted to integer representation. Integer underflow cannot occur.

Error Trapping

Error trapping allows you to trap and correct errors. Errors normally halt program execution and return error messages. When given before an error is encountered, the ON ERROR GOTO statement transfers program execution to a specified line number when an error occurs, and suppresses any error messages other than those you specify and write yourself. You can write an error-service routine at the specified location to identify and correct errors. In the following example, when −3 is entered in response to the INPUT statement, an error occurs and the user-written error message displays:

```
10 ON ERROR GOTO 100
20 INPUT A
30 PRINT SQR(A)
40 GOTO 20
100 PRINT "ARGUMENT CANNOT BE NEGATIVE."
110 PRINT "ENTER NEW VALUE."
120 RESUME NEXT
RUN
? 3
1.732051
? -3
ARGUMENT CANNOT BE NEGATIVE
ENTER NEW VALUE
?
```

Suppose there is no error routine (made up of lines 10, 100, 110, and 120) in the above example, and you wish to identify this particular error. The ERR and ERL functions return the error code number (see Appendix A) and the line number the error occurs in, respectively. In this way, you can identify the kind of error and the line it occurs in and write individual error handling routines for individual errors. The program below uses ERL and ERR to identify the location and nature of any error.

In the following example, the ON ERROR statement enables the error-service routine at line 100, which identifies one specific error (division by zero) and the line number in which it occurs:

```
10 ON ERROR GOTO 100
20 INPUT A
30 PRINT 52/A
40 GOTO 20
100 PRINT "Error";ERR;"at line";ERL
110 IF 11=ERR THEN PRINT "Division by zero"
120 RESUME NEXT
RUN
? 13
  4
? 0
Error 11 at line 30
Division by zero
?
```

Trace Facility

The TRON and TROFF (trace on and trace off) commands are used to examine the execution of each line. The line number of each program statement executed is displayed in brackets. The preceding example would run like this after entering TRON and then RUN:

```
[10] [20]? ■ 5 (CR)
[30] 10.4
[40] [20]? ■ 7 (CR)
[30] 7.428571
[40] [20]? ■ 0 (CR)
[30] [100] Error 11 at line 30
[110] Division by zero
[120] [40] [20]? ■
```

The ON ERROR GOTO 0 statement has a special meaning. When entered outside of error handling routines it disables any prior ON ERROR GOTO so that errors are handled normally—error messages are displayed and execution is halted. When entered inside an error handling routine, it allows normal handling of a given error or errors, as shown in the example below.

```
10 ON ERROR GOTO 100
20 INPUT A,B
30 PRINT A/B
40 PRINT SQR(B-A)
50 STOP
100 IF ERR<>11 THEN 130
110 B=1
120 RESUME
130 ON ERROR GOTO 0
RUN
? 2,0
  2
Illegal function call in 40
Ok
```

The error routine (lines 100-130) sets B to 1 when the divide-by-zero error occurs, but allows other errors to interrupt operation in a conventional way.

The non-fatal floating-point overflow and divide-by-zero errors are trapped like other errors. No messages are displayed in this case, and the results of the calculations are undefined.

Error Simulation

BASIC-80 provides a statement that can simulate any errors which produce an error code. When the ERROR statement is encountered during program execution, BASIC-80 acts as if the specified error had occurred. The example below demonstrates how ERROR can be used to test an error handling routine.

```
10 ON ERROR GOTO 70
20 INPUT A
30 B = A*.3842
40 PRINT B
50 GOTO 20
70 IF 11 = ERR THEN PRINT "Division by zero"
80 RESUME NEXT
```

If line 40 is replaced with:

```
40 ERROR 11
```

Program control transfers to line 70 regardless of the result of the operation in line 30.

Restarting Program Execution

You can restart program execution after it has halted with the RUN or CONT instruction, and after an error in an ON ERROR routine with the RESUME instruction. Remember that RUN sets all variables to zero.

RUN restarts program execution at the lowest-numbered line of a program if no line number is specified, or at the specified line number. It can be used at any time.

CONT restarts execution at the line after a STOP, END, or CONTROL-C is encountered. CONT will also restart program execution after an error, but it will try to execute the erroneous line.

RESUME can only be used when the halt occurs in an ON ERROR routine. It is given following the error-resolving routine pointed to by the ON ERROR instruction. It can be used three ways: RESUME to begin execution at the erroneous line; RESUME NEXT to begin execution at the line following the erroneous line; RESUME followed by a line number to begin execution at any other line.

CHAPTER 5
DISK FILE INPUT/OUTPUT

BASIC-80 includes two types of disk file Input/Output operations: sequential and random. Sequential I/O lets you read or write a file from the beginning to the end; random I/O lets you specify where in the file you read or write.

Because BASIC-80 runs under ISIS-II and RMX/80, filenames correspond to the ISIS-II and RMX/80-DFS format (:Fn:name.ext, where n is the drive number). BASIC-80 also gives you access to disk file-handling commands:

- DIR, which lists the files on a disk
- RENAME, which changes the name of a disk file
- ATTRIB, which changes the attributes of a disk file
- KILL, which deletes a disk file (it is actually the system DELETE command, but the name must be different because BASIC-80 includes a DELETE command to delete lines from a program).

For more information about ISIS-II filenames or operations, see the *ISIS-II User's Guide.*

Although both sequential and random I/O allow you to create, read, and write files on disk, sequential I/O is somewhat simpler in concepts and operation. If you haven't worked with disk files before, it would probably be better to start with sequential I/O to learn the principles.

Sequential File I/O

BASIC-80 Sequential I/O allows you to build sequential data files containing numbers and strings of up to 255 characters. In general, to use sequential I/O you must open the file, execute a series of INPUT (to read) or PRINT (to write) statements, then close the file.

A sequential file is open either for input or output. To switch from one to the other, you must close the file and open it again with the opposite attribute. Any I/O operation that immediately follows an OPEN statement starts at the beginning of the file.

Opening a Sequential File

You open a sequential file with the OPEN statement. It specifies whether the file is to be opened for input or output, assigns it a file number, and specifies the filename. Up to six files can be open at one time. If the file named in an OPEN statement that specifies output does not exist, it is created. Once a file is open, it cannot be opened again without closing it first. If you attempt to do so, an error message results.

To open an input file named :F1:DATES and assign it file number 1:

```
OPEN "I",#1,":F1:DATES"
```

String and numeric expressions can replace any of the parameters:

```
OPEN M$,FN, DN$ + "OUTPUT"
```

This statement opens a file for either input or output, depending on the value of M$ (it must be either "I" or "O") and assigns the file number represented by FN (it must be from 1 to 6). The name of the file is the value of the string variable DN$ (it should be ":F0:" through ":F9:", depending on how many drives you have, or null) followed by the six characters OUTPUT.

NOTE

Remember that opening a file for output destroys an existing file of the same name.

Writing to a Sequential File

After you open a sequential file for output, you can write data to it with the PRINT statement. If you open an existing file for output, its contents are deleted. To specify that a PRINT statement is being used to write to a sequential disk file, the file number (preceded by #, followed by a comma) follows the word PRINT.

To print a series of constants and variables to file 3:

```
PRINT #3, "Today's date is ";MO$;DA;",";YR
```

You can also use the full formatting capabilities of the PRINT USING statement:

```
PRINT #3 USING, "###,##;"234.41;81.20;4.68
```

Refer to Chapter 6 for further details of PRINT USING.

Reading from a Sequential File

After you open a sequential file for input, you can read data from it with the INPUT and LINE INPUT statements. To specify that these statements are being used to read from a sequential disk file, the file number (preceded by # and followed by a comma) follows the word INPUT or LINE INPUT.

The INPUT statement reads the specified number of values (numeric or string) from the disk file. Numeric values (in the disk file) can be separated by a blank, comma, carriage return, or line feed.

String values can be enclosed in quotation marks, or given as unquoted strings. Quoted strings terminate on the quote marks, while unquoted strings terminate on commas, carriage returns, line feeds, or if they exceed 255 characters in length.

Assume, for example, that disk file #2 contains the following data (a new line represents a carriage return-line feed pair in the disk file):

```
"R1",200,"R2",2200,"R3",10000
"R4",.47
```

After the file is opened for input, the following INPUT statement would assign a string value to R$ and the subsequent numeric value to R:

```
INPUT #2,R$,R
```

If executed four times, it would read all eight values.

LINE INPUT, on the other hand, ignores blanks, commas, and quotation marks, and reads everything between the current file position and the next carriage return-line feed, or up to 255 characters. If the last INPUT did not read to the end of a logical line, LINE INPUT will read to the end of that line before reading the next line. The string is assigned to the single string variable specified. In the preceding example, assume that instead of the INPUT statement shown, the following statement was executed:

```
LINE INPUT #2,R$
```

The value of R$ would be:

```
"R1",200,"R2",2200,"R3",10000
```

This statement is useful for reading from a file that consists of lines, not individual data values, such as lines from an ASCII-saved BASIC-80 program file, or a document created with the ISIS-II text editor.

The INPUT$ function reads the specified number of characters from a sequential file:

```
10 OPEN "I",#1,":F1:MONTHS"
20 PRINT INPUT$ (3,1)
30 CLOSE #1
40 END
```

This program reads the first 3 characters from the file :F1:MONTHS and prints them. If you execute line 20 again, INPUT$ will read the next 3 characters.

When you read the last value from a file, an end-of-file flag is set to −1. This flag can be tested with the EOF function. If you try to read data from a disk file after the end-of-file flag is set, BASIC-80 issues an INPUT PAST END message and execution halts. To prevent surprises, you can test for an end-of-file with the EOF function and then branch to an appropriate routine:

```
250 IF EOF (1) THEN 300
 .
 .
 .
300 CLOSE 1
310 PRINT "END OF INPUT DATA"
 .
 .
 .
```

Closing a Sequential File

Once you read or write the desired data from or to a sequential file, you must close the file. This can be done in several ways.

The CLOSE command closes one or more files:

```
CLOSE 1,2,5
```

If you enter CLOSE with no file number, all open files are closed. The END statement and the NEW and EXIT commands close all disk files. After closing a file, you can open it again for input or output, with the same or a different file number.

Random File I/O

Random I/O requires a bit more care and coding than sequential I/O, but is more flexible because you can move back and forth within a disk file. These are the key differences:

- To get data from random disk files to BASIC-80 variables and vice-versa, you must understand how BASIC-80 uses I/O buffer space.
- Random I/O operations (GET and PUT) always read or write all 128 bytes of a floppy disk sector. You must specify how this 128-byte string is divided into fields and assigned to string variables.
- Because all data in random files is treated as string data, you must convert strings that represent numbers into numeric values if you want to do any calculations with them. Likewise, if you want to write a numeric value to a random file, you must first convert it to a string. Special functions are provided to do this.
- You must use special assignment statements to put a string variable into an I/O buffer. This is due to the fact that the fields you define in an I/O buffer are of fixed-length; the special assignment statements accommodate variables that are longer or shorter than the specified field.
- Unless you simply want to read or write records in sequence, you must specify which record you want to read or write.

I/O Buffers

BASIC-80 reserves space for six I/O buffers, each 128 bytes long. One buffer is assigned to each open file (when an OPEN statement is executed). When you read a record from a random file using GET, the 128-byte string is not assigned directly to a variable. It is placed in the I/O buffer assigned to that file, and it is up to you to define what portions of the string are to be assigned to what variable or variables.

To emphasize: the usual variable and line delimiters (", CR) have no special meaning. All 128 bytes from the specified sector are placed in the I/O buffer. If fewer than 128 bytes were written to the sector, the remaining character positions in the buffer are set to ASCII blanks (hexadecimal 20).

Defining a Random I/O Field—FIELD

When a sector has been read into the I/O buffer, it exists as a 128-byte string. You must define how the string is to be assigned to a variable or variables. The FIELD statement specifies to BASIC-80 what character positions in the buffer correspond to string variables (initially they *must* be string variables, because the sector is treated as a 128-byte string).

The FIELD statement tells BASIC-80, for a given file, what character positions, starting with the first, are to be associated with a variable. For example, assume you have opened a random file, assigned it the file number 3 (the OPEN command works just like in sequential I/O), and read the first sector from the file. You know that the first 20 characters of each sector contain a name. To assign these 20 bytes to the string variable N$, enter the following FIELD statement:

```
FIELD #3, 20 AS N$
```

The FIELD statement can precede or follow the GET statement, but N$ doesn't contain those first 20 bytes until the FIELD statement. (A similar definition of fields must be done when writing to a random file with the PUT statement; the FIELD statement, obviously, must precede the PUT statement so that BASIC-80 will know which variable or variables to write to the disk.)

Any number of fields can be defined with one FIELD statement. In the preceding example, for example, let's assume that character positions 21-29 (the 9 bytes following the name field) contain a social security number. To define the first 20 bytes as N$ and the next 9 bytes as SS$, enter the following FIELD statement:

```
FIELD #3, 20 AS N$, 9 AS SS$
```

Not only can you define all 128 bytes this way, you can enter more than one FIELD statement for the same buffer. If more than one FIELD statement associates a string variable with character positions in the buffer, the last-executed one applies. If several FIELD statements are executed that specify different string variables, however, all the field definitions are effective.

Figure 5-1 summarizes this relationship of disk record, I/O buffer, and string variables, and shows the sequence of BASIC-80 statements necessary to read a record, assign the first 20 bytes to N$, and the next 9 bytes to SS$ (the remaining 99 bytes are ASCII blanks). The program shown opens the file, reads the field record, and prints both N$ and SS$.

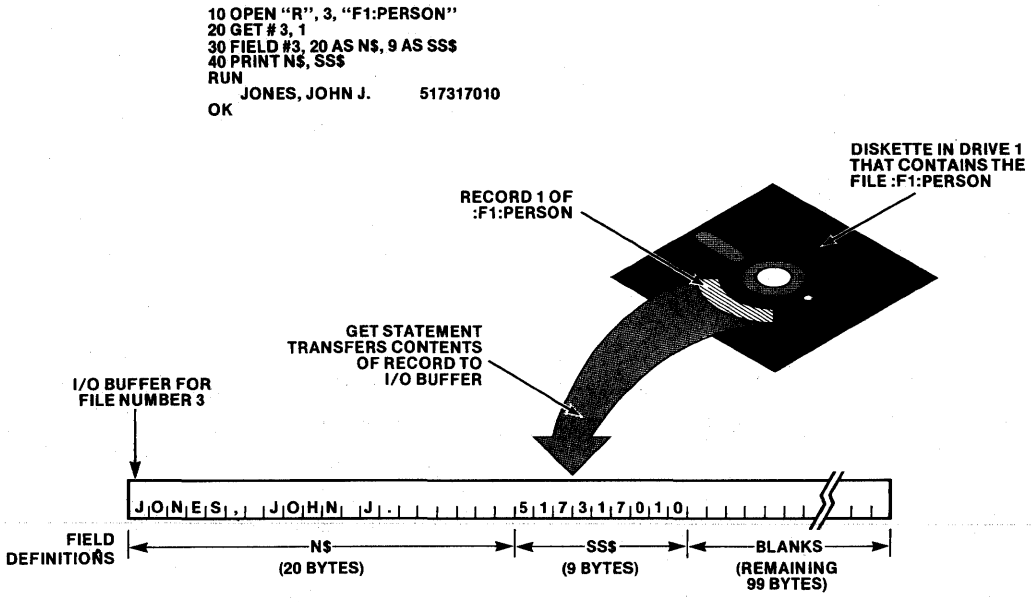

Figure 5-1. Random I/O Characteristics

Opening and Closing a Random Disk File

As with sequential disk files, you must open random disk files before you can write to them or read from them, and close them when you're through. A random file isn't opened for either input or output, however; once it's open for random I/O, you can read from it or write to it interchangeably.

To open the file ":F1:PERSON" for random I/O and assign it file number 3:

```
OPEN "R", #3, ":F1:PERSON"
```

Any of the parameters can be variables:

```
OPEN "R", F, FN$
```

This opens the file whose name is represented by the string variable FN$ for random I/O and assigns it the file number represented by the numeric variable F.

As with sequential I/O, the CLOSE statement can specify a file number (or numbers); CLOSE by itself closes all open files. The END statement and the NEW and EXIT commands close all open files; STOP does not.

Reading from a Random I/O File

You can retrieve data formatted and stored on disk with the GET statement. When entering the GET statement, you must specify the file you want to read. You can also specify the record you wish to read by entering the record number after the file number. If you don't specify the record number, BASIC-80 either reads the first record, if there has been no read operation since the file was opened, or reads the record that follows the last record read. To read the first record from file 1:

```
GET #3, 1
```

To read the next record:

```
GET #3
```

As described earlier (in "I/O Buffers" and "Defining Random I/O Fields—FIELD), the GET statement reads 128 bytes into the I/O buffer assigned to the random file by the OPEN statement. The FIELD statement specifies which portions of the buffer are to be assigned to string variables (all data in random I/O files is stored as string data).

Assume, for example, you have a file written as a random I/O file named :F1:PERSON. The first 20 characters of each sector contains a name, and the next 9 characters contain a social security number. The following statements open the file, define which parts of the buffer correspond to variables, and read the first sector:

```
10 OPEN "R", #3, ":F1:PERSON"
20 FIELD #3, 20 AS N$, 9 AS SS$
30 GET #3, 1
```

To print the name:

```
40 PRINT N$
```

It's not quite so simple to print or do calculations with numeric values from random I/O files, however, because they are represented as strings. You must convert each string that represents a number to its corresponding numeric value, either integer, single-precision floating-point, or double-precision floating point. The CVI, CVS, and CVD functions, respectively, perform this conversion.

Assume that there is an additional 2-character field following the 9-character social security number that represents the number of hours worked in the previous week. The following statements open the file, define the fields, read the first (or next) record, and increment the variable HW by the number of hours worked:

```
10 OPEN "R", #3, ":F1:PERSON"
20 FIELD #3, 20 AS N$, 9 AS SS$, 2 AS H$
30 GET #3
40 PRINT N$; SS$; "HOURS ="; CVI (H$)
50 HW = HW + CVI (H$)
60 PRINT "TOTAL HOURS ="; HW
```

The two characters defined as H$ are converted to an integer. To convert a string to a numeric variable, the number of string characters must equal the number of bytes required to store the corresponding numeric data type:

Integer	CVI	2 bytes
Single-precision value	CVS	4 bytes
Double-precision value	CVD	8 bytes

If the number of string characters represented by the argument of the function is fewer than required, an ILLEGAL FUNCTION CALL error message is printed and execution halts. If the string variable is longer than required, the extra characters are ignored.

Writing to a Random I/O File

To store data in a random disk file, you follow the same sequence of OPEN and FIELD statements as when reading from a random disk file. The PUT statement writes the 128 bytes of the I/O buffer to the disk file.

The FIELD statement, however, defines fixed-length fields in the I/O buffer. To get the data to be written to disk into the I/O buffer, there are two special assignment statements: LSET and RSET. LSET left-aligns a variable in the buffer field, and RSET right-aligns it. In both cases, if the variable is shorter than the field, it is padded with ASCII blanks (20H); if the variable is longer than the field, it is truncated.

For example, assume you have opened a file for random I/O, assigned it file number 3, and defined the first 20 characters as N$. You must use LSET or RSET to place values in the buffer:

```
10 OPEN "R", #3, ":F1:PERSON"
20 FIELD #3, 20 AS N$
30 LSET N$ = A$
40 PUT #3, 1
```

This technique may be used for storing variables for use in another program, linked to the first program with the RUN statement (see description of RUN, chapter 6).

Whatever the length of A$, its leftmost 20 characters are now identified as N$. If A$ is less than 20 characters long, the remaining characters of N are set to ASCII blanks (20H). Statement 40 writes all 128 bytes of the I/O buffer to disk file number #3.

Remember that random disk files are stored as strings. Just as you must convert fields that represent numeric values when you read them (using CVI, CVS, or CVD), so must you convert numeric values to their corresponding strings when writing to a random file, using the MKI$, MKS$, and MKD$ functions. Again, you must be sure to allow for the number of bytes required to store the three numeric data types.

Integer	MKI$	2 bytes
Single-precision value	MKS$	4 bytes
Double-precision value	MKD$	8 bytes

For example, assume that you open a random file and define two fields: the first 2 characters as P$ and the next 4 characters as C$; the first field represents a part number and the second its cost. The following statements open the file, set P$ equal to a variable PA and C$ equal to a variable CO, then write the two values to file number 5:

```
10 OPEN "R", #5, ":F1:PARTS"
20 FIELD #5, 2 AS P$, 4 AS C$
30 LSET P$ = MKI$ (PA)
40 RSET C$ = MKS$ (CO)
50 PUT #5, 1
```

The part number, of course, must be between −32768 and 32767, and the cost will be accurate to 7 digits.

Two other functions are related to random disk I/O: LOC and LOF. LOF returns the number of the last sector in the file; LOC returns the number of the sector that was last read or written.

CHAPTER 6
COMMANDS AND STATEMENTS

ATTRIB

The ATTRIB command changes certain specified attributes of BASIC-80 disk files. You can protect any file from writing or deleting or renaming by enabling the 'write protect' attribute, and you can later disable this attribute to alter or delete the specified file. The formats for these two operations are given below.

You can also enable or disable other file attributes with the ATTRIB command. Refer to the *ISIS-II User's Guide, File Control Commands,* for information on using ATTRIB with system, format, and invisible attributes.

To protect a file from writing, deletion, and renaming:

```
ATTRIB ":Fdrive number:filename", "W1"
```

To write to, delete, or rename a previously protected file:

```
ATTRIB ":Fdrive number:filename", "W0"
```

AUTO

The AUTO command automatically generates a line number at the beginning of each line when you are entering a program. The first operand specifies the number of the first line entered after the AUTO command; the second, which is optional, specifies the increment. Both operands default to 10.

```
AUTO [first line][, increment]
```

To number lines starting with 10 in increments of 10:

```
AUTO
```

To number lines with an increment of 50 starting with line 300:

```
AUTO 300, 50
```

If AUTO generates a line number that already exists in the program, the generated number is followed by an asterisk. To erase the existing line, enter CR; to leave that line in the program, enter Control-C to stop the AUTO process.

CLEAR

The CLEAR command sets all variables, arrays, and function definitions to zero, and closes files. If variables are referenced after a CLEAR, numeric variables will be zero and string variables will be null. The first expression is specified as an argument, and the value of this expression becomes the string space, in bytes.

```
CLEAR [expression[,address]]
```

If a second expression is specified, it indicates the highest memory location available to BASIC. This number is similar to the MEMTOP option. If not specified, the highest memory location used is unchanged. At sign-on only 100 bytes of string storage space are available.

```
CLEAR 2000, 0D000H
```

will reserve 2000 (decimal) bytes for string space, and will not allow BASIC to use any memory address above 0D000H.

CLOSE

The CLOSE statement concludes I/O activities for one or more disk files. A CLOSE statement for an open file disassociates the file name and the file number used to open the file, but the file may be opened again under the same or another file number. Close without options closes all files.

The END, NEW, MERGE, LOAD, RUN (without F) and CLOSE (without arguments) statements close all disk files automatically, but STOP does not.

```
CLOSE [[#] file number [,[#] file number]...]
```

```
10 OPEN "R",#3, "F1:PERSON"
20 GET #3,1
30 FIELD #3,20,AS N$, 9 AS SS$
40 PRINT N$,SS$
50 CLOSE #3
RUN
   JONES, JOHN J.        517317010
Ok
```

CONT

The CONT command continues program execution after a Control-C, a STOP, an ERROR interruption, or an END statement has been executed. If execution halts because of an error, BASIC-80 will try to re-execute the same line again. Execution resumes at the statement after the break occurred unless input from the terminal was interrupted; in this case, the prompt (?) is reprinted. Execution cannot be continued if the program was modified during the break, but variable values may be changed.

```
CONT
```

In the example below, the program expects a numeric input and is interrupted with a Control-C.

```
ENTER NEXT INTEGER STRING
? 41 ↑C
BREAK IN 240
OK
CONT
? 37
.
.
.
```

DATA

The DATA statement prefaces lines of data that are read sequentially by the READ command and assigned as values to variables. The data is separated by commas and may be numeric, strings, or quoted strings. If strings are not surrounded by quotes (" ") they may not contain commas (,) or semicolons (;) and leading or trailing blanks are ignored. DATA instruction lines may be located anywhere in program text. BASIC maintains a pointer which keeps track of the next DATA element to be read. This pointer is set to the first data element of the first DATA statement when a program is RUN. It increments to subsequent DATA elements when its elements are read. It can be moved with the RESTORE command.

```
DATA number|string literal|quoted string
 [, number|string literal|quoted string]...
```

```
10 DATA 10, IS LESS THAN, 77
20 DATA 44, IS GREATER THAN, -32
30 DATA 1. 7, "IS EQUAL TO ", 1.7E0
40 FOR I=1 TO 3
50 READ X, A$, Y
60 PRINT X; A$; Y
70 NEXT
RUN
10 IS LESS THAN 77
44 IS GREATER THAN -32
1.7 IS EQUAL TO 1.7
Ok
```

DEF FN(X)

The DEF FN(X) statement defines arithmetic or string functions for later use in program text. The name given to the function must be FN followed by a valid variable name. The variable(s) given are the arguments used within the function. Any one-line expression can be used. User-defined functions can be of any type, arithmetic or string, and any number of arguments are allowed. The correct number of parameters must be specified when the function is referenced within a program. Functions may not be redefined.

```
DEF function name [(variable [, variable...])]= expression
```

The rules for function name are the same as for variable name.

```
10 DEF FNAC (X) = (1/(2*3.14159))*X
20 FOR X = 1 TO 10
30 PRINT FNAC (X),
40 NEXT X
50 END
RUN
.159155    .31831     .477465    .63662      .795776
.954931   1.11409    1.27324    1.4324      1.59155
```

DEFSNG
DEFDBL
DEFSTR
DEFINT

The DEFSNG, DEFDBL, DEFSTR, and DEFINT statements are used to specify that a given block of letters will designate a specific data type when used as the first letter of variable names. The DEFSNG, DEFDBL, DEFSTR, and DEFINT statements specify single precision floating point, double precision floating point, string, and integer representation, respectively. Blocks of letters are specified according to the syntactic format given below:

```
DEFXXX m [-n][,m[-n]]...
```

in which m represents any letter, A through Z, and n represents any letter B through Z. These two letters represent the boundaries of a block of letters between m and n. If n is not used, only variables beginning with letter m are specified. The n argument must follow m in alphabetic sequence; in other words, blocks may be defined as D to L, or A to G, but not as P to A or Z to M.

```
10 DEFSTR S
20 DEFDBL D
30 DEFINT I-N
.
.
.
```

DEFUSRn

The DEFUSRn statement is used to specify the starting address of an 8080/8085 assembly-language, PL/M-80, or Fortran-80 subroutine so they may be called by the USR function. The argument n may be an integer from 0 to 24; if no argument is given, 0 is assumed. Two examples of DEFUSRn statements are given below:

```
DEFUSR = 0F000H
DEFUSR7 = I%
```

Because of the way BASIC-80 represents numbers internally, 32767 is the largest value for an integer expression. The following code can be used to specify higher decimal values in a DEFUSRn statement.

```
10 INPUT "ENTER ENTRY ADDRESS IN DECIMAL"; A
20 IF A > 32767 THEN A = A-65536
30 DEFUSR11 = A
```

The program below shows how DEFUSR defines a subroutine. Refer to Appendix E for further details of using DEFUSR.

```
 10 REM This program asks for three
 20 REM Integers, passes them to USR1, and
 30 REM Returns the result.
 40 DEFINT A-Z
 50 DEFUSR 1 = 0E000H
 60 PRINT "Enter three numbers:"
 70 INPUT A, B, C
 80 PRINT "A = ";A; "B = ";B; "C = ";C
 90 RESULT = USR%1 (A, B, C)
100 PRINT "A + B + C + = " RESULT
110 END
```

DELETE

The DELETE command removes specified instruction lines from program text. A single line or block of lines may be deleted, as shown in the syntactic format and example below. Instruction lines prior to and including the specified line may be deleted by adding a dash (-) before a line number. When deleting a range of lines, the line numbers specified as end points must exist. If not, an error results.

DELETE line number|line number-line number|-line number

```
10 PRINT CHR$(12)
20 PRINT:PRINT:PRINT
30 REM THIS PROGRAM FINDS THE AVERAGE
40 REM OF THREE NUMBERS
50 INPUT A,B,C
60 PRINT (A+B+C)/3
70 END
DELETE -40
OK
LIST
50 INPUT A,B,C
60 PRINT (A+B+C)/3
70 END
OK
DELETE 70
OK
LIST
50 INPUT A,B,C
60 PRINT (A+B+C)/3
Ok
```

DIM

The DIM statement defines the number of elements in an array variable. If an array variable is not dimensioned before it is referenced, a default value of 10 is assumed as the maximum possible subscript range for each dimension in the reference. If given, DIM statements must allocate space before you refer to the arrays they dimension, since once an array is dimensioned, these dimensions cannot be changed. If an OPTION BASE n statement is used in the program to specify starting points for arrays, it should precede the first DIM statement, or array reference.

DIM variable (integer [,integer]...) [,variable (integer [,integer]...)...]

In the example below, A is a single precision array with 21 elements, indexed with a value of from 0 to 20. I is an integer array with 13 times 14 (= 182) elements. S$ is a three dimensional string array with 125 elements. DD# is a singly dimensioned double precision array with 22 elements.

```
10 DEFINT I-N
20 DIM A (20), I (12,13), S$ (4,4,4), DD# (21)
.
.
.
```

DIR

The DIR command displays the names, number of blocks, and length (in bytes) of the files saved on the specified disk. If no drive number is specified, the default is drive 0. DIR may be used as a command or as a statement.

```
DIR  [drive number]
```

EDIT

The EDIT command is used to modify single program lines. In this mode, you have a selection of editing subcommands that facilitate character insertion, deletion and changing. These subcommands and their uses are explained in Chapter 3: Entering and Editing BASIC-80.

The EDIT mode is entered automatically when a syntax error prevents program execution. It can also be entered when entering text by typing Control A, or it may be entered by typing EDIT line number. Refer to Chapter 3: Entering and Editing BASIC-80.

```
EDIT line number|.
```

The period (.) is used to represent the last line changed, listed or that contained an error.

END

The END statement halts program execution and returns to BASIC-80 command level. Files are closed, but variables can be examined. END may be used at any logical conclusion point within program text. STOP acts like END, but also prints a "BREAK IN line number" message, and does not close files. An END command is optional as the last line of a program.

```
END
```

The program below contains an END statement in line 40.

```
10 INPUT A,B,C
20 D=(A+B+C)/3
30 PRINT "THE AVERAGE OF";A;" + ";B;" + ";C;"EQUALS";D;"."
40 END
```

ERROR

The ERROR statement simulates the occurrence of an error in program execution. The user-supplied integer expression will generate an error message appropriate to its value, as shown in the example listed below. The integer expression supplied must be greater than 0 and less than or equal to 255. If ERROR is executed with a number that does not correspond to a BASIC error message, "UNPRINTABLE ERROR" will print. All errors can be trapped by the ON ERROR statement. For further information, refer to Chapter 4: Error Handling.

ERROR expression

```
10 INPUT A,B,C
12 IF B=0 THEN ERROR 11
14 PRINT A/B
20 END
RUN
? 40, 0,17
DIVISION BY ZERO IN 12
Ok
```

EXIT

The EXIT command terminates operation of the BASIC-80 interpreter and returns control to ISIS-II. EXIT closes all open files, but does not automatically save programs. If you EXIT without saving the current program, it will be lost.

EXIT

FIELD

The FIELD statement is used to allocate space within one of six 128-byte BASIC-80 random file buffers. When a random I/O transaction occurs, data move to or from the specified random file buffer to or from disk sectors. The FIELD statement associates the data with its position in the I/O buffer so that future references to the specified variable will access the proper string characters or numbers within the record stored in the random file buffer. The example below shows how FIELD is used. For further information, refer to the BASIC-80 Disk Input/Output section.

FIELD # file number,

number of characters AS string variable

[,number of characters AS string variable]...

The program below opens random file :F1:FILE.1 and asks for five 30-character strings to insert into :F1:FILE.1.

```
10 OPEN "R",#1,":F1:FILE.1"
20 FIELD #1, 30 AS A$
30 FOR I = 1 to 5
40 INPUT Z$
50 LSET A$ = Z$
60 PUT #1,I
70 NEXT I
80 CLOSE #1
90 END
```

FOR-NEXT-STEP

The FOR-NEXT statement allows the execution of a given group of lines a given number of times. As shown in the example below, on each iteration of the routine, value A is incremented by the value given in the STEP instruction. If no value for STEP is specified, the increment is 1. The instruction lines following the line containing the FOR instruction are executed until the NEXT instruction is encountered. Program control returns to the FOR instruction line, STEP is added to the value of A, and the routine is reentered. When variable A exceeds the TO expression, control passes to the instruction line following the NEXT instruction line.

If the TO expression is larger than the starting expression and the step expression is negative then the loop will not execute. Similarly, if the TO expression is less than the starting expression and the STEP expression is positive, then the loop will not execute. After the conclusion of the loop, variable A contains the first value not used in the loop.

Control cannot be passed to a statement within a FOR...NEXT loop with GO TO, GO SUB etc, but a GO TO can be used to jump out of a FOR...NEXT loop. FOR...NEXT loops may be nested so long as each inner loop is completely contained in all outer loops, and they employ different index variables.

If possible, integer index variables should be used to speed execution. Double precision index variables are not allowed.

FOR variable=expression TO expression [STEP expression]

```
10 FOR A = 1 TO 5
20 PRINT A*A,
30 NEXT A
40 END
RUN
1     4     9     16     25
```

GET

The GET statement reads the specified sector from the specified file into 128- byte random buffer. The file must be open for random access. If the sector number is not specified, the sector number of the previous GET or PUT is incremented and this sector is read. A GET without a sector number immediately after a file is opened will return the first sector. The largest possible sector number is 2046. If the GET command is issued for a sector that has not yet been written, buffer contents are undefined, and there is no error message. However, if no greater record has ever been PUT or GET, the file will be extended.

GET [#] file number [,record number]

```
10 OPEN "R",#3, "F1:PERSON"
20 GET #3, 1
30 FIELD #3, 20 AS N$, 9 AS SS$
40 PRINT N$,SS$
RUN
  JONES, JOHN J.          517317010
Ok
```

GOSUB

The GOSUB statement transfers program control to the line number specified as an argument. This line number is typically the first line number of a subroutine that is used several times during a program. The RETURN statement transfers control back to the first instruction that follows the last GOSUB statement executed.

GOSUB line number

```
 10 INPUT A
 20 IF A>0 THEN GOSUB 100 ELSE PRINT "A MUST BE >0"
 30 GOTO 10
100 PRINT SQR(A0);SIN(A);TAN(A)
110 RETURN
```

GOTO

The GOTO (line number) statement transfers execution of instruction lines from the current line to the line number specified. There is no provision for a return to the branching point.

GOTO line number| GO TO line number

The GO TO in line 270 causes a jump to line 100.

```
.
.
.
250 IF EOF(6) THEN 300
260 PRINT "FILE STILL CONTAINS DATA"
270 GOTO 100
300 PRINT "FILE ";F$;"IS OUT OF DATA."
310 END
```

IF-THEN-ELSE

The IF-THEN-ELSE statement evaluates the given logical expression and executes the THEN instruction if true, the ELSE instruction if it is false. The ELSE instruction is optional; if it is left out, and the logical expression is false, the next line is executed. Line numbers can be used in place of instructions and have the same effect as GO TOs. An IF statement must always be on one line. IF statements may also be nested, as shown in the example below. In this example, an ELSE clause matches the last unmatched THEN.

IF expression THEN instruction [ELSE instruction]

```
 5 IF S$ = "  " THEN 40
10 IF S$ = "+" THEN A = A+B ELSE IF S$ = "-" THEN A=A-B ELSE 30
20 GOTO 40
30 PRINT"Invalid S$"
40 END
```

INPUT

The INPUT statement returns a prompt (question mark) and waits for the user to enter data. String literals or numeric values may be expected, according to the type of variable specified as an argument.

INPUT [quoted string;] variable [,variable]...

If a string expression follows INPUT and is followed by a ";" it is used in place of the question mark as the prompt. If the number or type of data items in the response do not agree with the number or type of variables requested, BASIC-80 returns a prompt to re-enter the line. Strings which are input need not be surrounded by quotes as long as they do not contain commas (,). Leading and trailing blanks are ignored in this mode.

```
10 INPUT A,B,C
20 PRINT (A+B+C)/5
30 END
RUN
? 40,2,18
12
Ok
```

KILL

The KILL command deletes files from disk storage, and removes all references to the deleted file from the directory. Once a file has been killed, it cannot be reopened.

```
KILL filename
```

LET

The LET statement is used to give a value to a variable. If a variable appears to the left of an equal sign not preceded by the word LET, BASIC-80 assumes that a LET statement is implied. When assigning a literal value to a string variable, the value must be enclosed in quotation marks.

```
[LET] variable = expression
```

```
10 LET A=5
20 PRINT A
RUN
5
Ok
```

LINE INPUT

The LINE INPUT statement reads an entire line of data and assigns it as the value of the specified string variable. LINE INPUT does this in two ways: first, data may be read from the terminal and assigned; second, data may be read from a disk file and assigned.

When entering data from a terminal, the syntactic format is:

```
LINE INPUT [string expression;] string variable
```

This statement will print the string expression as a prompt and will halt until the string is entered and a carriage return has been entered. Entering a Control-C will abort LINE INPUT and return BASIC-80 to the Command Mode. A CONT command may then be used to re-enter the LINE INPUT statement.

When entering data from a disk file, the syntactic format used is:

```
LINE INPUT #file number,string variable
```

This statement reads data from the specified disk file until a carriage return is encountered. The data is assigned as a value for the specified string variable, and LINE INPUT skips over the encountered carriage return-line feed to point to the beginning of the next string. If no carriage return is encountered, the statement returns the next 255 characters of data. See Chapter 5 for further information about disk file I/O.

LIST

The LIST command prints the current program's text to the console. Instruction lines are listed in sequential order. As shown below, the LIST command may be used to display the entire program, all instruction lines up to or after a given line, or a block of instruction lines within the program. The period may be used as an abbreviation for the last line which was changed, contained an error, or was listed.

```
LIST [line number| -line number| line number-| line number-line number]
```

In the example below, LIST 30 prints line 30 only; LIST -30 prints all lines up to and including 30; list 30- prints all lines after 30, including 30; LIST 30-50 prints lines 30, 40, and 50; LIST prints the entire program text.

```
10 PRINT CHR$(12)
20 INPUT A,B,C
30 PRINT (A+B+C)/3
40 PRINT
50 END
LIST 30
30 PRINT (A+B+C)/3
LIST-30
10 PRINT CHR$(12)
20 INPUT A,B,C
30 PRINT (A+B+C)/3
LIST 30-
30 PRINT (A+B+C)/3
40 PRINT
50 END
LIST 30-50
30 PRINT (A+B+C)/3
40 PRINT
50 END
LIST
10 PRINT CHR$(12)
20 INPUT A,B,C
30 PRINT (A+B+C)/3
40 PRINT
50 END
```

To obtain a listing of a file on a line printer or other device, use the SAVE command, which is described on page 6-22.

LOAD

The LOAD command reads a file from disk and stores it in memory, making it the current file. The string expression must be an ISIS-II filename.

```
LOAD filename
```

```
LOAD ":F1: PLOT"
OK
RUN

THIS PROGRAM PLOTS ANY USER-DEFINED
FUNCTION ON AN X-Y GRAPH.
.
.
.


.
.
.
```

LSET, RSET

The LSET and RSET statements store the specified string expression into string space that has been allocated in a random file buffer by a FIELD statement. When string variables are assigned to a random file buffer with the FIELD statement, they

cannot later be assigned values with the LET statement. This is because strings are assigned new storage locations when given new values and this destroys the effect of the FIELD statement. Accordingly, the LSET and RSET statements must be used to assign new values. These commands may also be used with normal string statements. If the assigned string value is shorter than the space reserved for the string variable in the FIELD statement, LSET left justifies the string, adding blanks (ASCII 20H) as needed. RSET right justifies the string in the same way. If the string is longer than the space reserved, the extra characters are ignored. LSET and RSET may be used with string variables that have not been fielded. Unlike normal string assignment, the length of the receiving string is not changed by an LSET or RSET assignment.

```
LSET string variable = string expression
RSET string variable = string expression
```

```
10 A$ = "1 2 3 4 5 6"
20 RSET A$ = "A B"
30 PRINT A$; "X"
40 LSET A$ = "C D"
50 PRINT A$; "X"
RUN
     ABX
CD     X
Ok
```

MERGE

The MERGE command reads a program from disk and combines it with the current program without changing line numbers of either program. If the new program and the current program have instruction lines with the same line numbers, the new instruction lines replace the current ones. The new program must have been saved in ASCII format. MERGE sets all variables and arrays to zero. If used within a program, the program will end. A given string expression must resolve to an ISIS-II filename.

```
MERGE string expression
```

```
MERGE ":F4: EVAL.BAS"
Ok
```

NEW

The NEW command deletes the current program and clears all variables and arrays. If a program has not been saved and the NEW command is given, the program is lost.

```
NEW
```

In the sample program below, the program has been deleted by the NEW command. The LIST command will not list it, or any program, until another is read from a peripheral device or entered line-by-line.

```
240 PRINT "end of program"
250 PRINT
260 PRINT
270 END
NEW
Ok
LIST
Ok
```

NEXT

The NEXT statement is used with a previous FOR statement to end an iteration of a FOR-NEXT loop; when BASIC-80 encounters a NEXT statement, control passes back to the statement line containing the last FOR statement. If no index variable is specified, BASIC-80 increments the variable specified in the last FOR statement. Each NEXT can end more than one loop if the index variables used in each loop are given separated by commas.

The syntax of NEXT is:

```
NEXT [Variable] [, Variable]...
```

```
10 FOR A=1 TO 5
20 PRINT A*A;
30 NEXT A
40 END
RUN
1  4  9  16  25
Ok
```

NULL

The NULL command specifies the number of null characters printed at the end of a printed line, following the carriage return. This feature is used with hard copy terminals that require a certain number of null characters that set carriage return timing.

```
NULL number of null characters to be transmitted
```

ON ERROR GOTO

The ON ERROR GOTO statement transfers program control to the specified line number when error conditions occur or an ERROR instruction simulates an error within a program. The ERR function is set to the applicable error code and the ERL function is set to the applicable line number.

The instruction line ON ERROR GOTO 0 removes the effect of any previous ON ERROR GOTO, so that errors cause messages to print and halt execution. If ON ERROR GOTO 0 is executed within an error-handling routine (after an error but before a RESUME) the proper error message prints and execution halts immediately.

If an error occurs in an error routine, after an ON ERROR GOTO branch and before a RESUME, the ON ERROR GOTO has no effect. This means that if an ON ERROR GOTO is in effect, and two errors (or ERROR commands) occur without an intervening RESUME, the second error prints an error message and halt execution.

```
ON ERROR GOTO line number
```

```
10 ON ERROR GOTO 200
20 ERROR 11
.
.
.
200 PRINT "ERROR" ERR "AT LINE" ERL
210 RESUME NEXT
```

ON ... GOSUB

The ON...GOSUB statement transfers program control to one or one of a set of subroutines.The line numbers of the first lines of these subroutines follow sequentially, separated by commas. If the expression evaluates to 3, control transfers to the third line number following GOSUB.

```
ON expression GOSUB line number [,line number] ...

10 INPUT A
20 ON A GOSUB 200, 300
```

ON...GOTO

The ON...GOTO statement transfers program control to one or one of a set of line numbers. The expression X is evaluated and control is transferred to the "nth" line number following the line containing the ON...GOTO instruction. If the expression evaluates to 3, control transfers to the third line number following GOSUB.

```
ON expression GOTO line number [,line number]...

10 INPUT A
20 ON A GOTO 500,510,520
.
.
.
```

OPEN

The OPEN statement makes an ISIS-II file available to BASIC. It also associates a file number from 1 to 6 with the file. File type can be I, O, or R. Random I/O files are specified by an R, sequential input files are specified by an I, and sequential output files are specified by an O, as shown in the syntactic format below. The file number is an integer expression greater than 0 and less than or èqual to 6, is preceded by a "#" sign, and is used to reference the file in I/O transactions.

Only six files may be open at one time. Note that SAVE, LOAD, DSKF, DIR and MERGE all require a file, so a maximum of five files may be open before they are executed.

BASIC-80 buffers files, so if :CI: is opened for input, BASIC-80 will wait for the first line of console input.

```
OPEN type, [#] file number, filename
```

where type is a string expression equal to "O", "I" or "R". In the example below, the file SYSLIB is opened as a random I/O file, with number 4.

```
OPEN "R",#4,"SYSLIB"
```

The example below shows a typical use of OPEN in an I/O program.See Chapter 5 for further details of disk random I/O.

```
10 OPEN "R",#3, "F1:PERSON"
20 GET #3,1
30 FIELD #3,20 AS N$, 9 AS SS$
40 PRINT N$,SS$
RUN
     JONES, JOHN J.   517317010
Ok
```

OPTION BASE

The OPTION BASE command is used to begin indexing of arrays at 1 or 0. By specifying an argument of 0 or 1, you can begin all arrays at 0 or 1. If present, OPTION BASE should precede all DIM statements and array references. If OPTION BASE 1 is not specified, 0 is the default value.

```
OPTION BASE 0|1
```

OUT

The OUT statement writes the specified value to the specified I/O port. The value is an integer expression in the range 0 to 255.

```
OUT port number, expression
```

This example rings the bell on Intellec development systems using the RS-232 port.

```
OUT 0F6H, 7
```

POKE

The POKE statement places the specified value into the memory location specified. The specified value is an expression that must round to an integer in the range 0-255. POKE can also accept hexadecimal and octal arguments, even though octal and hex constants over 32767 (decimal) are interpreted by BASIC-80 as negative numbers. It does this by adding 65536 to such arguments.

```
POKE location, expression

POKE 0FFFFH, 0
POKE 65535, 0
POKE 177777Q, 0
POKE -1, 0
```

The instructions above set the top byte of memory to 0 in an Intellec development system having 64K of memory. Before using POKE, you should use the CLEAR command to reserve free memory. A POKE into BASIC or system code may cause BASIC or other software to fail.

PRINT

The PRINT statement returns the value of expressions or the text of strings to the console, or to any ISIS-II file. Literal strings must be enclosed in quotation marks (“); variables and expressions need no quotation marks.

You can print data to any ISIS-II file by specifying a file number. If a comma (,) is inserted following an expression to be printed, the next printing starts at the beginning of the next 14-column zone. If a semicolon (;) is inserted, the next printing starts at the next column. If no punctuation is used at the end of the PRINT statement, the next printing starts at the beginning of the next line. PRINT by itself prints one blank line. A “?” entered after a line number means PRINT.

```
PRINT [# filenumber,][[ expression ],|;]...
```

```
10 INPUT "A string"; X$
20 PRINT "YOUR STRING IS ";X$
30 END
RUN
? A string
YOUR STRING IS A string
Ok
```

PRINT USING

The PRINT USING statement specifies particular formats to print strings or numbers. The format string may include more than one field, and may also include any literal character that is not a PRINT USING formatting character. The syntactic format of PRINT USING is:

```
PRINT [# file number,] USING format string; expression [, expression]...
```

in which the format string specifies spacing and additional characters (such as asterisks and dollar signs) when used with numeric fields, and which specify portions of the given string when used with string fields. Each expression is printed with the same format. The optional file number specifies the number of an open file. See Chapter 5 for details of sequential disk file I/O.

String Fields

There are two formatting choices when using PRINT USING with string fields:

“!” specifies that the first character of each specified string is printed.

“\ n spaces \” specifies that the first 2+N characters are printed. If the backslashes are used without spaces, two characters are printed.

If the string field has more characters than the format string, the balance are ignored; if there are fewer characters than specified, spaces are printed to fill out the field. Here are examples of the string field formats:

```
10 X$="ONE"
20 Y$="TWO"
30 PRINT USING "!"; X$; Y$
40 PRINT USING" \ \"; X$; Y$
RUN
OT
ONE TWO
Ok
```

As can be seen, PRINT USING "!" prints a string consisting of the first letters of each string. PRINT USING \ 2 spaces \ prints a string consisting of four characters from X$ (3 plus a space) and four from Y$ (also 3 plus a space). If Y$="SEVEN", the result printed when line 40 is executed would be:

```
ONE SEVE
```

and if X$="SEVEN" and Y$="EIGHT", the results of line 40 would be:

```
SEVEEIGH
```

Numeric fields

The PRINT USING formatting characters specify right justification, leading dollar signs, plus or minus signs, exponentiation, asterisk fills, or insertion of commas. Each of the BASIC-80 format characters is explained in the following section.

The number or sharp sign (#) indicates the position of digits. If the number to be printed is too small to fill all of the digit positions specified to the left of the decimal point, leading spaces print to fill the entire field. An integer can print as up to five digits, single precision floating-point up to seven digits, and double precision floating-point up to fifteen digits.

. The decimal point can be inserted in any position in the field. If the format specifies that a digit follows the decimal point, the digit always prints, whether it is a 0 or another number. Numbers are rounded as necessary.

```
PRINT USING "##.##"; 41.287
41.29

PRINT USING "##.##"; 71/100
0.71
```

+ The plus sign (+) is used at the beginning or end of the field of format characters. Insertion of this character prints a plus or minus sign, depending on the sign of the number, at the beginning or end of the number, respectively.

− The minus sign (−) is used at the right of the format character field to force printing of a trailing minus sign if the number is negative. If it is positive, a space is printed. If neither (+) or (−) is included in the format, a negative number is printed with a leading (−).

```
PRINT USING "+#.##"; 4.89; -2.6689
+4.89    -2.67

PRINT USING "##.#-"; -5.8; 96.2
5.8-     96.2
```

****** The double asterisk (**) fills any leading blank spaces in the number format with asterisks. This symbol also specifies positions for two more digits.

```
PRINT USING "**##.#"; 4.8; 243.3
***4.8    *243.3
```

$$ The double dollar sign ($$) adds a single dollar sign to the immediate left of the number being formatted.

```
PRINT USING "$$###.##"; -48.28; 364.90
-$48.28     $364.90
```

($$) specifies space for two additional characters, but the $ added takes up one position. The exponential format cannot be used with ($$).

$ The double asterisk-dollar sign ($) returns the results of both the (**) and ($$) format characters. Exponential format cannot be used; "**$" allows for three additional digit positions, one of which is the dollar sign.

The comma (,) is placed to the left of a decimal point to print a comma to the left of every third digit on the left of the decimal point. The comma also specifies another digit position. A comma to the right of the decimal point prints in that position. The comma cannot be used with exponentiation.

```
PRINT USING "#####,.##"; 92114.84
92, 114.84
```

↑↑↑↑ or ΛΛΛ Four carat signs (or vertical arrows) specifying exponentation. The carats or arrows are placed after the numeric format characters. Any decimal point format may be used. The significant digits are left justified, and the exponent is adjusted. The comma and floating ($) cannot be used when exponentiation is specified.

% The percent (%) character indicates that a number larger than the given format has been encountered. BASIC-80 returns the number itself preceded by a (%) sign; or, if rounding the number causes it to exceed the specified field, the rounded number is printed, preceded by a (%) sign. If the number of specified digits exceeds 24, an ILLEGAL FUNCTION CALL error results.

```
PRINT USING "##.##"; 40.48; 766784; 99997
40.48% 766784.00% 99997.00
```

PRUN

The PRUN command starts execution of a program stored in PROM. The address of the program is an integer argument. The program must be saved in ASCII, and followed by a (Control-Z).

```
PRUN address
```

PUT

The PUT statement transfers data from the previously formatted random file buffer to the specified disk sector. The PUT statement requires the file number assigned when the random file was opened and the sector the data goes to. If no sector number is specified, and no data have been written to the disk, the first disk sector is written to. If data have been written to the disk, the number of the last sector written increments, and the next sector is written to. The sector number may not exceed 2048.

```
PUT file number, [sector number]

10 OPEN "R", 1, ":F1:FILE.1"
20 FIELD 10 AS A$
30 INPUT A$
40 PUT 1,1
50 CLOSE 1
60 STOP
RUN
? "A STRING"
Ok
```

RANDOMIZE

The RANDOMIZE statement prompts the user for a new random number seed for the random number function RND. If the RANDOMIZE command is not given within a program, the same sequence of random numbers will repeat each time any program is run.

```
RANDOMIZE [(expression)]
```

If the optional expression is included, its value becomes the new seed and no prompt is issued.

```
10 RANDOMIZE
20 A = RND
30 IF A < .5 THEN PRINT "HEADS"
40 IF A >= .5 THEN PRINT "TAILS"
RUN
Random number seed (0-65529)? 123
TAILS
Ok
```

READ

The READ statement sequentially assigns values found in the accompanying DATA statements to the variables specified. If more DATA fields are read than are available, an OUT OF DATA error message will be printed. If there is still data available to be read, the next READ statement will assign the next datum to the specified variable. The READ pointer may be reset to the first item of any DATA statement with the RESTORE command. Any type of variable may be read as long as the type of the DATA is appropriate for the type of variable.

```
READ variable [,variable]...

10 DATA 2,4,6,8,10,12,14,16,18,20
20 READ A,B,C,
30 PRINT A;B;C
40 READ D,E,F
50 PRINT D;E;F
RUN
2 4 6
8 10 12
Ok
```

REM

The REM statement is used to insert commentary remarks into program text. Any instruction line that begins with REM following its line number is passed over, and program control passes to the next line. Within a remark, : (colon) is simply another character, not a statement separator.

```
REM

10 REM THIS PROGRAM FINDS THE AVERAGE
20 REM OF THREE NUMBERS
30 INPUT A,B,C
40 PRINT (A + B + C)/3
50 END
```

RENAME

The RENAME command changes the name of the specified file to the new filename. Only the directory reference (accessed with the DIR command) is altered; if programs reference the old filename, a FILE NOT FOUND error message will result.

```
RENAME "old filename" TO "new filename"
```

RENUM

The RENUM command resequences line numbers in a program, whether they appear at the beginning of a line or as the object of a GOTO statement. You can specify up to three optional arguments: the first is the new number of the first line to be renumbered, the second is the old number of the first line to be renumbered, and the third is the increment between lines. If you specify no arguments, the new number of the first line to be renumbered is assumed to be 10, all lines are renumbered, and the increment is assumed to be 10.

```
RENUM [new number][,[old number][,increment]]

20 INPUT A
40 PRINT "NEW:"; A
15 RANDOMIZE
 5 PRINT CHR$(12)
52 A1 = A*RND
58 A2 = INT(A1)
RENUM
LIST
10 PRINT CHR$(12)
20 RANDOMIZE
30 INPUT A
40 PRINT "NEW:"; A
50 A1 = A*RND
60 A2 = INT(A1)
```

RESTORE

The RESTORE statement resets the READ pointer to the first DATA statement within program text. The DATA statements can then be reread. A line number can be specified following RESTORE; if included, data on the specified line will be read next.

```
RESTORE [line number]

.
.
10 DATA 48,49,51,53,58
20 DATA 104,108,116,132,164,5000,5000
30 READ A,B
.
.
560 IF B<5000 THEN 30
570 RESTORE
580 READ A,B
.
.
```

RESUME

The RESUME statement restarts program execution after an error has been detected and handled. Program execution begins at the line specified; or if no line number is specified, execution resumes at the statement where the error occurred. The statement RESUME NEXT causes execution at the statement following the erroneous statement.

```
RESUME [line number|NEXT]

.
.
140 ON ERROR GOTO 200
.
.
200 PRINT "ERROR";ERR;"AT LINE";ERL;"."
210 PRINT
220 RESUME NEXT
.
.
```

RETURN

The RETURN statement is placed at the end of a subroutine, and transfers program control back to the instruction line immediately following the last GOSUB or ON... GOSUB statement executed.

```
RETURN

.
.
50 GOSUB 200
60 PRINT "TYPE NEXT NUMBER"
.
.
200 IF A>100 THEN PRINT "OUT OF RANGE"
210 IF A<=0 THEN PRINT "A MUST BE >0"
220 A=1/(B*B1)
230 PRINT
240 PRINT "YOUR NEW FIGURE IS ";A;
```

RUN

The RUN command starts the execution of a program or a set of programs. If you enter RUN alone, it executes the current program starting at the lowest line number. If you enter RUN followed by a line number, it executes the current program starting at the specified line number. If you enter RUN followed by a string variable representing a file name, it looks for a file with that name on disk, loads its contents into memory, and executes the program starting at the lowest line number. With the latter form, an "F" may be added to leave files open. Otherwise, they are closed. Files can be left open only if the program was saved in non-ASCII format. The RUN command erases all variables before executing the program.

```
RUN [line number|string expression [,F]]
```

The RUN command may also be used as a program statement to chain programs together. The form of the RUN command that performs this function is:

```
line number RUN"filename"
```

This feature is especially useful for the execution of programs that are larger than can fit into memory as one large program. The RUN command initializes all variables to zero, however, so if you want to pass some variables to the next program, print them out to a buffer file (see the section on "Random Disk I/O" for details) during execution of the first program. Then read them, when needed, into subsequent programs.

In the example below, the (↑C) character halts program execution.

```
LIST
10 INPUT A,B,C
20 PRINT (A+B+C+)/3
30 PRINT"NEXT SERIES"
40 PRINT : PRINT : PRINT
50 GOTO 10
60 END
RUN
? 41, 12, 6
19.6
NEXT SERIES

? ↑C (This control character interrupts execution)
BREAK IN 10
OK
RUN 30
NEXT SERIES

?
```

SAVE

The SAVE command stores the current program on disk with the specified filename. The addition of a comma and an A saves the file in ASCII format. Note that if a file with the specified name exists, it will be replaced by the new file. Any valid ISIS-II file name may be used. You can use a string expression that resolves to an ISIS-II filename. For example, SAVE ":LP:", A will list the current program on the line printer. Refer to the *ISIS-II User's Guide* for more information on ISIS- II device filenames.

```
SAVE string expression [,A]

10 INPUT A,B,C
20 PRINT (A + B + C)/3
30 END
SAVE "AVER"
OK
RUN "AVER"
? 5,8,2
5
Ok
```

STOP

The STOP statement halts program execution and prints a BREAK IN (line number) message. Following this, BASIC-80 is in the command mode. After execution of a STOP, program variables may be changed or printed, and, if the program is not changed, execution may be resumed with the CONT command. The END statement also halts execution, but does not print a BREAK IN (line number) message, and closes any open file.

```
STOP

.
.
.
320 PRINT "END OF PROGRAM"
330 STOP
RUN
END OF PROGRAM
BREAK IN 330
Ok
```

SWAP

The SWAP statement exchanges the values assigned to any two variables or string variables, provided that they are of the same type.

```
SWAP variable1,variable2

.
.
.
110 IF X>Y THEN SWAP X,Y
120 PRINT "NEW VALUE OF X IS";X;""AND Y IS NOW";Y
.
.
.
```

TRON, TROFF

The two program tracing commands, TRON and TROFF, respectively enable and disable the trace function in program execution. When enabled, the number of each line is printed as it is executed, enclosed in square brackets. These commands can also be used within programs to selectively enable and disable tracing.

```
TRON
TROFF

10 INPUT A,B,C
20 PRINT (A+B+C)/3
30 END
TRON
OK
RUN
[10]42, 48, 45
[20]45
[30]
OK
TROFF
OK
RUN
30, 18, 12
20
Ok
```

WAIT

The WAIT statement instructs BASIC-80 to monitor incoming bytes from a specified port. These bytes are tested with a mask byte, which is an integer expression in the range 0 to 255. The resulting bits are compared with the corresponding bits of the comparison byte, also an integer expression in the range or 0 to 255. If the comparison byte is not specified, the default value is 0. Execution of program text is halted until the two sets of bits differ. When the WAIT command is given, the port status is exclusive ORed with the comparison byte and the result is ANDed with the mask byte. Execution halts until a non-zero value is obtained.

```
WAIT port number, mask byte,[comparison byte]
```

WIDTH

The WIDTH command adjusts the width of the lines printed at the console to the specified value, which is an expression in the range 15 to 255. The default value for WIDTH is 72.

```
WIDTH expression

10 A$ = "ACTION, OHIO; WEAVERTON, VERMONT; HOLLOWAY,
CALIFORNIA"
20 PRINT A$
30 WIDTH 15
40 PRINT A$
50 WIDTH 72
60 END
RUN
ACTION, OHIO; WEAVERTON, VERMONT; HOLLOWAY, CALIFORNIA
ACTION, OHIO; WE
AVERTON, VERMON
T; HOLLOWAY, CA
LIFORNIA
Ok
```

CHAPTER 7
FUNCTIONS

ABS

The ABS function returns the absolute value of the specified expression. The absolute value of an expression is its numeric value with a positive sign.

```
ABS(expression)

10 INPUT A
20 A = ABS(A*2)
30 PRINT A
40 END
RUN
? 5
10
Ok
RUN
? -5
10
Ok
```

ASC

The ASC function returns the ASCII code of the first character of the specified string.

```
ASC(string expression)

PRINT ASC ("0"), ASC("AB")
48          65
```

ATN

The ATN function returns the arctangent value of the specified expression. The result, given in radians, is between −PI/2 and PI/2. The calculation is performed in single precision, but the argument may be any numeric type.

```
ATN(expression)

10 INPUT A
20 B = ATN(A)
30 PRINT B
40 END
RUN
? 5
1.373401
Ok
```

CDBL

The CDBL(X) function changes the type of expression (X) into double precision floating-point representation. In this format, calculations are accurate to 16 decimal places, compared to an accuracy of 7 decimal places in single precision floating-point representation.

Many fractions (such as 1/3 and .1) cannot be precisely represented with either single or double precision floating point, due to BASIC-80's internal numeric representation.

```
CDBL(expression)
```

```
10 A# = 1/3
20 B# = CDBL (1)/3
30 PRINT A#,B#
40 END
RUN
.333333343267441              .333333333333333
Ok
```

CHR$

The CHR$ function evaluates the expression to an integer and returns the equivalent ASCII character. The expression must evaluate to an integer between 0 and 255.

```
CHR$ (expression)
```

In the program below, the CHR$(12) command generates a form feed in line 110.

```
.
.
.
110 PRINT CHR$(12)
120 PRINT
130 PRINT "TABLE 2-5: COMMONLY USED CONSTANTS"
140 PRINT
150 PRINT A;B;C;D;E
.
.
.
```

CINT

The CINT(X) function rounds the expression into an integer value and returns that value. If the expression is less than -32768 or greater than + 32767 then overflow will result. NOTE: The FIX and INT functions also convert to integer Formats. FIX truncates; INT truncates to an integer argument.

```
CINT(expression)
```

This program averages three numeric values and returns an integer result:

```
10 INPUT A,B,C
20 PRINT (CINT((A + B + C)/3))
30 END
RUN
? 45, 24, 77
49
```

COS

The COS function returns the cosine value of the specified expression. The input is given in radians. The calculation is performed in single precision.

COS(expression)

```
10 INPUT A
20 B = COS(A)
30 PRINT B
40 END
RUN
? 5
.2836621
Ok
```

CSNG

The CSNG(X) function changes the type of expression into single-precision floating-point representation. In this format, calculations are accurate to seven decimal places, compared to an accuracy of sixteen decimal places in double precision floating-point representation.

Performing calculations in single precision is faster than in double precision but may not be as accurate.

CSNG(expression)

```
10 A# = 1.D0/3.D0
20 PRINT CSNG (A#) /2, A#/2
RUN
.1666667      .1666666666666667
```

CVI
CVS
CVD

The CVI, CVS, and CVD functions convert strings into numbers. CVI converts a two character string into an integer by simply moving the two characters into the internal integer representation. CVS converts a four character string to single precision, and CVD converts an eight character string into double precision in a similar fashion.

These functions are used to retrieve numeric values from a random file buffer field, and are the reverse of MKI$, MKS$, and MKD$.

CVx(string expression)

```
A% = 1
B! = 2
C# = 3
D$ = MKI$(A%) + MKS$(B!) + MKD$(C#)
A% = CVI (D$)
B! = CVS (MID$(D$, 3))
C# = CVD (MID$(D$, 7))
PRINT A, B!, C#
      1       2       3
```

DSKF

The DSKF function returns the number of 128-byte sectors that are free on the specified disk. The examples below signify that there are 50*128, or 6.4 kilobytes of free space on disk 1.

```
DSKF   (drive number)

PRINT DSKF (1)
50
OK
```

EOF

The EOF function detects when the end of the file is reached when reading a sequential data file. If there are no more data in the file, EOF returns a -1, indicating TRUE. At any other time, EOF will return 0, indicating FALSE. If the file is not open, an error results.

```
EOF (file number)

10 OPEN "I", 1, "[F1: PROGRAM]
20 IF EOF(1) THEN 60
30 LINE INPUT #1, A$
40 PRINT A$
50 GO TO 20
60 END
```

ERL

The ERL function returns the number of the line in which the last error took place. It is normally used after an ON ERROR GOTO...error trapping routine.

```
ERL
```

The program below uses the error-trapping instruction in line 10 to print the error description in line 40. This line also contains the function ERR, which displays the code number of the error. In this example the error is division by 0, which has the error code 11:

```
10 ON ERROR GOTO 40
20 A = 1/0
30 END
40 PRINT "PROGRAM ENDS IN LINE"ERL"WITH ERROR"ERR
50 RESUME NEXT
RUN
PROGRAM ENDS IN LINE 20 WITH ERROR 11
Ok
```

ERR

The ERR function returns the code number of the last error encountered during program execution.

```
ERR
```

The following program asks for a divisor and a dividend, and uses the error-trapping routine in lines 60-90 to prevent division by zero (which has code 11) from stopping the execution:

```
10 ON ERROR GOTO 60
20 INPUT "WHAT ARE THE NUMBERS TO DIVIDE";X,Y
30 Z = X/Y
40 PRINT "QUOTIENT IS";Z
50 GOTO 20
60 IF ERR <> 11 OR ERL <> 30 THEN 90
70 PRINT "YOU CAN'T HAVE A DIVISOR OF ZERO!"
80 RESUME 20
90 ON ERROR GOTO 0
100 END
RUN
WHAT ARE THE NUMBERS TO DIVIDE?
?045,9
QUOTIENT IS 5
WHAT ARE THE NUMBERS TO DIVIDE? 11,0
? 11,0
YOU CAN'T HAVE A DIVISOR OF ZERO!
WHAT ARE THE NUMBERS TO DIVIDE?
?
```

EXP

The function EXP returns the result of raising e (2.71828) to the specified power. The specified value must be less than 88.7, or overflow occurs. The calculation is performed in single precision.

```
EXP(expression)
```

```
10 INPUT A
20 B = EXP(A)
30 PRINT B
40 END
RUN
? 5
148.4131
OK
?
```

FIX

The FIX function truncates numbers with decimal fractions to their integer value.

```
FIX (number)
```

```
PRINT FIX (41.99999)
41
Ok
```

FRE

The FRE function returns the number of bytes of memory left when the dummy argument (X) is given. If you give a string variable as a dummy argument (X$), FRE returns the number of free bytes in string storage space.

```
FRE (X)|(X$)
```

```
PRINT FRE(X)
22490
Ok
```

HEX$

The HEX$(X) function returns a string of hexadecimal digits which represents the hexadecimal value of the integer argument. In BASIC-80, integers are from −32768 to 32767, but 8080/8085 memory addresses go from 0 to 65535 decimal. HEX$ will handle arguments in both ranges correctly.

```
HEX$ (expression)

PRINT HEX$ (−1), HEX$ (65535)
```

Returns:

```
FFFF  FFFF
```

```
HEX$ (expression)

10 PRINT "TYPE THE NUMBER OF BYTES OF SPACE"
20 PRINT "NEEDED FOR YOUR PROGRAM, IN DECIMAL:"
30 PRINT
40 INPUT A
50 A$ = HEX$(A)
60 PRINT "YOU'LL NEED "; A$; "BYTES"
```

INP

The INP function returns the value of a byte from the input port specified by the expression. The expression must be an integer expression in the range 0-255.

```
INP (expression)

.
.
.
50 IF INP(A) = 12 THEN PRINT "FORM FEED"
60 IF INP(A) = 7  THEN PRINT "BELL"
.
.
.
```

INPUT$

The INPUT$ function reads to a specified number of characters from a specified sequential file. In the example below you can type in any ten or more characters from the console into sequential file #2. The :CI: specifies console input. Line 20 prints the first 10 characters of file #2, and CLOSE concludes the reading of the data. INPUT$ treats Carriage Return like any other character.

```
INPUT$ (characters, filenumber)

10 OPEN "I", #2, ":CI:"
20 PRINT INPUT$ (10,2)
30 CLOSE #2
40 END
RUN
1234567890   (CR)
1234567890
```

INSTR

The INSTR function searches for the first occurrence of the second given string within the first given string, and returns the first position of the second string as an ordinal number. The optional argument, an expression I greater than 0 and less than 255, starts the search at I characters. The INSTR function returns a 0 under three conditions: if I is greater than the length of the first string, if the second string cannot be found in the first string, or if the first string contains no characters.

INSTR ([I,] string expression, string expression)

```
10 A$ = "RANDOM NUMBER SUBROUTINE"
20 B$ = "R"
30 PRINT INSTR(A$,B$)
40 PRINT INSTR(3,A$,B$)
50 END
RUN
1
13
Ok
```

INT

The INT function returns the largest integer value less than or equal to the specified expression. The sign of the returned value is the same as the sign of the specified expression.

INT(expression)

```
10 INPUT A
20 B = INT(A)
30 PRINT A
40 END
RUN
? 18.0427
18
OK
RUN
? -234.98
-235
```

LEFT$

The string function LEFT$(X$,I) evaluates the string X$ and returns the leftmost I characters. I is an integer in the range 0-255.

LEFT$ (string expression, expression)

```
10 X$ = "WHITE, SMITH, JONES, BLACK, GREEN"
20 Y$ = LEFT$(X$,11)
30 PRINT X$
40 PRINT Y$
RUN
WHITE, SMITH, JONES, BLACK, GREEN
WHITE, SMITH
Ok
```

LEN

The string function LEN(X$) returns the length, in number of characters, of string X$. All characters are counted, including non-printing characters and blanks.

LEN (string expression)

```
10 X$ = "JOHN J. JONES"
20 PRINT LEN(X$)
30 END
RUN
13
Ok
```

LOC

The LOC function has two uses. When used with a random file, LOC returns the current sector number. The current sector is defined as the last sector that was read or written. If no sectors have been read or written, LOC returns to 0. When used with a sequential file, LOC returns the number of sectors read or written to since the last OPEN statement was executed on that file.

LOC (file number)

```
10 OPEN "R", #3, :F1:RANDOM
20 GET #3, 44
30 PRINT LOC (3)
40 END
RUN
44
Ok
```

LOF

The LOF function returns the number of sectors in a random file. When LOF is used with a sequential file, it returns the number of data sectors (128 bytes per sector) in the file.

LOF (file number)

```
PRINT LOF(4)
33
```

LOG

The LOG function returns the natural logarithm of the argument. The calculation is performed in single precision.

LOG(expression)

```
10 INPUT A
20 B = LOG (A)
30 PRINT A
40 END
RUN
? 2488
7.81924
Ok
```

MID$

The MID$(X$,I.[,J]) function examines string X$ and returns the rightmost characters starting at pointer I. I and J are integers in the range 1-255. If argument J is specified, J characters are returned, starting at position I. If I is greater than LEN(X$), the MID$ function returns the null string. If argument J is greater than the number of characters in X$ to the right of I or is not specified, MID$ returns the rest of the string.

MID$(string expression, expression [,expression])

```
10 X$ = "JOHN J. JONES"
20 PRINT MID$(X$,10,3)
30 PRINT MID$(X$,9)
40 END
RUN
ONE
JONES
Ok
```

The MID$ (X$, I. [,J]) function may also appear on the left side of an assignment statement. Employed in this context, it will replace the characters of string X$ beginning at position I with the string given on the right. If J is specified , J characters of X$ are replaced. If I is greater then LEN(X$) , an illegal function call error is displayed. The length of X$ is never changed.

```
10 A$ = "ABCDEF"
20 B$ = "XXYYZZ"
30 MID$(A$,2,4) = B$
40 PRINT A$
50 END
RUN
AXXYYF
Ok
```

MKI$
MKS$
MKD$

The three functions MKI$, MKS$, and MKD$ convert data represented as numerical values into two-, four-, or eight-byte strings, respectively. MKI$ is used to convert an integer value; MKS$ is used to convert a single-precision floating-point value; and MKD$ is used to convert a double-precision floating-point value.

MKI$ (integer)
MKS$ (single-precision value)
MKD$ (double-precision value)

These functions are used to place numeric values into fields of random file buffers. See Chapter 5 for discussion of MKI$, MKD$, and MKS$

OCT$

The OCT$ function returns a string of octal digits which represent the value of the integer argument. The expression is rounded to an integer before conversion.

OCT$ (expression)

```
10 PRINT "TYPE DECIMAL INTEGER TO BE CONVERTED."
20 INPUT A
30 A$ = OCT$(A)
40 PRINT A,"EQUALS,"A$," IN OCTAL."
```

PEEK

The PEEK function reads a single byte from memory at the location specified. The corresponding POKE statement writes a byte into a specified memory location.

```
PEEK (expression)

PRINT PEEK(0FABH)
200
```

POS

The POS function returns the position of the cursor after the last PRINT statement. The argument I is a dummy argument. The leftmost position is 1.

```
POS (integer)

10 INPUT A$
20 PRINT A$;
30 IF POS(1) > 10 THEN PRINT
40 PRINT "HAS JUST BEEN INPUT"
RUN
?AAAAA
AAAAA HAS JUST BEEN INPUT
OK
RUN
? AAAAAAAAAA
AAAAAAAAAA
HAS JUST BEEN INPUT
Ok
```

RIGHT$

The RIGHT$ function returns the rightmost I characters of string X$. If I equals or exceeds the length of X$, the entire string is the result. If I is 0, a null string results.

```
RIGHT$ (string,integer)

10 A$ = "JOHN J. JONES"
20 X$ = RIGHT$(A$,8)
30 PRINT X$
RUN
J. JONES
Ok
```

RND

The RND function returns a single precision random number between 0 and 1. The sequence produced is identical every time a program is run. If this is undesirable (such as in games) use RANDOMIZE to prompt the user for a seed.

```
RND
```

SGN

The SGN function returns the sign of the specified expression. If the expression is greater than 0, SGN returns 1. If the expression is 0, SGN returns a 0. If the expression is less than 0, SGN returns -1.

SGN(expression)

```
10 INPUT A
20 LET B = 3.14159*SGN(A)
30 PRINT B
40 END
RUN
? 44
3.14159
OK
RUN
? -12
-3.14159
OK
RUN
? 0
0
Ok
```

SIN

The SIN function returns the sine value of the argument. The input is given in radians. The calculation is performed in single precision.

SIN(expression)

```
10 INPUT A
20 PRINT SIN(A)
30 END
RUN
? 8
.989358
Ok
```

SPACE$

The SPACE$ function returns a string of spaces equal to the value of the integer expression.

NOTE

SPACE$ returns an actual string.

SPACE$ (integer expression)

```
10 A = 1
20 PRINT "QUESTION";A;SPACE$(10);"THEORY"
30 PRINT
40 PRINT SPACE$(21);"TEXT"
50 END
RUN
QUESTION 1        THEORY

                  TEXT
```

SPC

The SPC function returns a string of spaces n characters long when used with a PRINT statement, as in the example below. SPC, unlike the SPACE$ function, does not return an actual string, only a series of spaces. It may only be used with a PRINT statement.

SPC (integer)

```
10 PRINT
20 PRINT SPC (10); "Question 1":PRINT:PRINT
30 PRINT SPC (15);"How many states are there in binary
     logic?":PRINT
40 INPUT A$:PRINT:PRINT
50 IF A$ = "2" THEN PRINT "Correct" ELSE GOTO 750
.
.
```

SQR

The SQR function returns the square root of the specified expression. The expression must evaluate to be greater than or equal to zero, or an error message is returned. SQR is calculated in single precision.

SQR(expression)

The program below finds the square root of input A, which is entered by the user, then displays it:

```
10 INPUT A
20 PRINT SQR(A)
30 END
RUN
? 45
6.70821
RUN
? -1
ILLEGAL FUNCTION CALL IN 20
```

STRING$

The STRING$ function returns a string of the same character repeated the specified number of times. If an integer argument is used, the ASCII character having that numeric code is returned the specified number of times. If a string argument is supplied, the first character of the string is returned the specified number of times.

STRING$ (expression, expression|string expression)

```
10  A$ = STRING$(10,97)
20  PRINT A$
RUN
aaaaaaaaaa
OK

10  A$ = STRING$(10,"A")
20  PRINT A$
RUN
AAAAAAAAAA
Ok
```

STR$

The STR$ function returns a string of decimal digits that represent the value of the integer expression.

STR$ (expression)

```
10  FOR I=0 TO 9
20  A$ = A$ + MID$ (STR$ (I), 2)
30  NEXT I
40  PRINT A$
RUN
0123456789
Ok
```

TAB

The TAB function spaces to the specified column position at the terminal. The leftmost column is 1 and the rightmost is the WIDTH value. If the current print position is beyond the specified column, TAB will force a carriage return, line feed before spacing to the specified column. This function may only be used with the PRINT statement. If the expression rounds to a value less than 1, TAB(1) results. If the expression rounds to over WIDTH value, TAB(expression MOD width) results.

TAB (expression)

```
10 FOR I= 1TO 4
20 PRINT TAB (I); I
30 NEXT I
RUN
1
        2
                3
                        4
Ok
```

TAN

The TAN function returns the tangent value of the argument. The input is given in radians. The calculation is performed in single precision.

TAN(expression)

```
10 INPUT A
20 PRINT TAN(A)
30 END
RUN
? 41
.160656
Ok
```

USRn

The USRn function is used to reference a user-defined assembly-language, PL/M-80 or Fortran-80 subroutine. The DEFUSRn statement specifies the starting address for the corresponding USRn subroutine. The argument n may be any integer from 0 to 24; if no argument is given, 0 is assumed. The type character indicates the type of the result. If none is indicated, the result must be returned as an integer in registers H and L.

Only integers can be used as arguments; other variables must be passed by reference. This is done with the VARPTR function, which returns the address of the specified variable.

```
USR[$|#|%|!][n][(parameter...)]
```

Here is an example of how the USRn statement is used:

```
10 CLEAR 1000, 0DFFFH
20 DEFUSR4 = 0E000H
30 A$ = "A STRING"
40 A = 1E4
50 A# = 14D-3
60 A% = 12
70 B = USR ! 4 (VARPTR(A!),VARPTR(K1),VARPTR(R#),VARPTR(LA%))
```

Arguments are returned in a similar fashion, unless the type character is omitted, in which case registers H and L are used.

Appendix E gives details of loading and running ASM-80, PL/M-80, and FORTRAN-80 subroutines that may be called with USRn.

VAL

The VAL function returns the numerical value of string X$. If the string does not represent a valid number, VAL(X$) equals 0.

```
VAL (string expression)
```

```
10 INPUT A$
20 IF VAL(A$) = 0 THEN 60
30 A1 = VAL(A$) * 52
40 PRINT A1, A$
50 END
60 PRINT "ENTER NUMERIC DATA ONLY."
70 GOTO 10
RUN
?4
 208      4
```

VARPTR

The VARPTR function returns the address in memory of a variable or the input/output buffer associated with a file number. If the variable has not yet been assigned a value, an ILLEGAL FUNCTION CALL error results. The main use of VARPTR is to pass variable or array addresses to assembly-language subroutines. Arrays are passed by specifying VARPTR (A (0)) (or VARPTR (A(1)) if OPTION BASE 1 is in effect) so that the lowest addressed element of the array is returned. All simple variables should be assigned values in a program before calling VARPTR for any array; otherwise, allocation of a new simple variable will cause the addresses of all arrays to change. See Appendix E for further information about using VARPTR.

```
VARPTR (variable|#file number)
```

APPENDIX A
BASIC-80 ERROR CODES

Table A-1. BASIC-80 Error Codes

Error	Description	Number
NEXT without FOR	Program contains no corresponding FOR for NEXT	1
SYNTAX ERROR	Illegal usage of delimiters, characters, etc.	2
RETURN without GOSUB	No GOSUB statement found to RETURN to.	3
Out of DATA	All DATA statements in the program have been read, or BASIC-80 tried to read too much, or too little data was included in the program.	4
Illegal function call	Parameter passed to a function was out of range. Possible reasons: 1. A negative array subscript 2. An array subscript > 32767 3. LOG with a zero or negative argument 4. SQR with a negative argument 5. A↑B with A negative and B not an integer 6. A call to USR before a corresponding DEFUSR 7. Calls to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACES$, INSTR, or ON...GOTO with improper arguments	5
Overflow	Variable with magnitude greater than 3.4E38 (single-precision floating-point) or 1.7D308 (double-precision floating-point)	6
Out of memory	Program too large, contains too many loops, subroutines, variables, or complicated expressions.	7
Undefined line number	A referenced line does not exist.	8
Subscript out of range	You referenced an array variable outside the dimensions of the array, or referenced the wrong number of dimensions.	9
Duplicate Definition	You redimensioned an array previously dimensioned.	10
Division by zero	A 'division by zero' operation was attempted.	11
Illegal direct	An instruction was used illegally in the Command Mode.	12
Type mismatch	A function which expects a string or variable was given the wrong data type; any improper mix of data types.	13
Out of string space	String variables are larger than the allocated space. You can increase space with CLEAR.	14
String too long	String exceeds 255 characters.	15
String formula too complex	String was too long or complex; break into two.	16
Can't continue	An attempt was made to continue a program where an error occurred, or after modifications, or with no program.	17
Undefined user function	Reference to a non-defined USR function.	18
No RESUME	An error trapping routine has no RESUME.	19
RESUME without error	RESUME statement without error-trapping routine.	20
Unprintable error	No error message exists for the given error. Check ERROR statements for undefined errors.	21

Table A-1. BASIC-80 Error Codes (Cont'd.)

Error	Description	Number
Missing operand	An operator was given without an operand.	22
Line buffer overflow	A program or data line has too many characters for the line buffer. Divide into two or more parts.	23
FOR without NEXT	A FOR statement was found without a corresponding NEXT	24
FIELD overflow	More than 128 characters were allocated in a FIELD statement.	50
Internal error	An error occurred in BASIC-80 internal execution. If this error cannot be accounted for, contact your Intel representative.	51
Bad file number	An unopened file was referenced.	52
File not found	A LOAD, KILL, or OPEN statement referenced a file not found on the specified disk.	53
Bad file mode	One of the following conditions apply: 1. The file mode is other than I, O, or R. 2. PUT or GET to a sequential file. 3. Opening a random file for sequential output, or vice versa. 4. Performing a PRINT to a random file.	54
File already open	An attempt to open an already opened file was made.	55
Disk I/O error	A disk I/O error has occurred on disk (x); this means a sector read check failed 18 times.	57
File already exists	File already exists.	58
Disk full	All disk space is full. Delete old files or try new disk.	61
Input past end	An INPUT statement has been given after the end-of-file; check INPUT operations with the EOF function.	62
Bad record number	The record number in a PUT or GET is > 2048 or < 0.	63
Bad file name	An invalid ISIS-II filename was given.	64
Direct statement in file	A direct statement was found while loading a program into BASIC-80. The LOAD is terminated.	66
Too many files	An attempt to open a new file after 6 files were opened.	67

APPENDIX B
BASIC-80 RESERVED WORDS

The following list shows 126 words that cannot be used as names of variables. If you attempt to do so, errors, error messages, or both will result. A valid variable name is one or more alphanumeric characters, the first of which must be a letter. If more than two characters are given, the rest are ignored.

ABS
AND
ASC
ATN
ATTRIB
AUTO
BASE
CDBL
CHR$
CINT
CLEAR
CLOSE
CONT
COS
CSNG
CVD
CVI
CVS
DATA
DEF
DEFDBL
DEFFN
DEFINT
DEFSNG
DEFSTR
DEFUSR
DELETE
DIM
DIR
DSKF
EDIT
ELSE
END
EOF
EQV
ERASE
ERL
ERR
ERROR
EXIT
EXP
FIELD
FIX
FN
FOR
FRE
GET
GOSUB
GOTO
HEX$
IF
IMP
INP
INPUT
INPUT$
INSTR
INT
KILL
LEFT
LEN
LET
LINE
LIST
LOAD
LOC
LOF
LOG
LSET
MERGE
MID$
MKD$
MKI$
MKS$
MOD
NEW
NEXT
NOT
NULL
OCT$
ON
OPEN
OPTION
OR
OUT
PEEK
POKE
POS
PRINT
PRUN
PUT
RANDOMIZE
READ
REM
RENAME
RENUM
RESUME
RESTORE
RETURN
RIGHT$
RND
RSET
RUN
SAVE
SGN
SIN
SPACES$
SPC
SQR
STEP
STOP
STR$
STRINGS$
SWAP
TAB
TAN
THEN
TO
TROFF
TRON
USING
USR
VAL
VARPTR
WAIT
WIDTH
XOR

APPENDIX C
BASIC-80 COMMAND CHARACTERS

BASIC-80 has certain single control characters (characters produced by pressing the letter and the CONTROL key simultaneously) that cause something to happen immediately. These characters are listed below.

To edit the last line entered:	CONTROL-A
To halt program execution and return to command level:	CONTROL-C
To tab across the line:	CONTROL-I
To resume program execution after it is stopped by Control-S:	CONTROL-Q
To halt program execution until Control-Q is entered:	CONTROL-S
To erase the current line:	CONTROL-X
To retype the current line:	CONTROL-R
To disable display to the terminal (until CONTROL-O is given again or the program runs to completion):	CONTROL-O

APPENDIX D
ASCII CODES

Table D-1. ASCII Code List

Decimal	Octal	Hexadecimal	Character
0	000	00	NUL
1	001	01	SOH
2	002	02	STX
3	003	03	ETX
4	004	04	EOT
5	005	05	ENQ
6	006	06	ACK
7	007	07	BEL
8	010	08	BS
9	011	09	HT
10	012	0A	LF
11	013	0B	VT
12	014	0C	FF
13	015	0D	CR
14	016	0E	SO
15	017	0F	SI
16	020	10	DLE
17	021	11	DC1
18	022	12	DC2
19	023	13	DC3
20	024	14	DC4
21	025	15	NAK
22	026	16	SYN
23	027	17	ETB
24	030	18	CAN
25	031	19	EM
26	032	1A	SUB
27	033	1B	ESC
28	034	1C	FS
29	035	1D	GS
30	036	1E	RS
31	037	1F	US
32	040	20	SP
33	041	21	!
34	042	22	"
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	'
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	,
45	055	2D	-
46	056	2E	.
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:
59	073	3B	;
60	074	3C	<
61	075	3D	=
62	076	3E	>
63	077	3F	?

Decimal	Octal	Hexadecimal	Character
64	100	40	@
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	^
95	137	5F	_
96	140	60	`
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~
127	177	7F	DEL

Table D-2. ASCII Code Definition

Abbreviation	Meaning	Decimal Code
NUL	NULL Character	0
SOH	Start of Heading	1
STX	Start of Text	2
ETX	End of Text	3
EOT	End of Transmission	4
ENQ	Enquiry	5
ACK	Acknowledge	6
BEL	Bell	7
BS	Backspace	8
HT	Horizontal Tabulation	9
LF	Line Feed	10
VT	Vertical Tabulation	11
FF	Form Feed	12
CR	Carriage Return	13
SO	Shift Out	14
SI	Shift In	15
DLE	Data Link Escape	16
DC1	Device Control 1	17
DC2	Device Control 2	18
DC3	Device Control 3	19
DC4	Device Control 4	20
NAK	Negative Acknowledge	21
SYN	Synchronous Idle	22
ETB	End of Transmission Block	23
CAN	Cancel	24
EM	End of Medium	25
SUB	Substitute	26
ESC	Escape	27
FS	File Separator	28
GS	Group Separator	29
RS	Record Separator	30
US	Unit Separator	31
SP	Space	32
DEL	Delete	127

APPENDIX E
CALLING NON-BASIC-80 SUBROUTINES

You can write a subroutine in FORTRAN-80, PL/M-80, or 8080/8085 assembly language, convert it into relocatable code, load it into free memory, and access it directly from BASIC-80. Any number of variables can be referenced, following PL/M conventions for passing parameters to subroutines.

You will need the *ISIS-II User's Guide* and the publication relevant to the language you use for the subroutine. The preface lists the Programming and Operator's manuals for FORTRAN-80, PL/M-80, and 8080/8085 assembly language.

Preparing Subroutines

Once you have written the desired subroutine, follow the instructions in the appropriate compiler or Assembler Operator's Manual to generate object code from your source language.

The compiler or assembler output is a relocatable object code. This code is given a starting address in Intellec system memory. To do this, you must know the highest starting address you can use, as well as the total free memory space.

A 48K Intellec system has the highest usable address of 0BEBFH. A 64K system has a highest usable address of 0F6BFH. Higher-addressed memory in both systems is taken by the monitor and/or monitor RAM.

When you invoke BASIC-80, it immediately returns the free memory space in bytes. The size of your main program and subroutine(s) must be less than the free space.

Suppose you have a 64K system, and a 2K byte (800H) subroutine. The highest usable memory address is 0F6BFH. If you place your program next to the monitor, it must start at 0EEC0H. You must be sure that there is 2K of space available at 0EEC0H, and you must forbid BASIC-80 to use this space with the MEMTOP option when you invoke BASIC-80. MEMTOP specifies the highest RAM address BASIC-80 may use. In the example below, MEMTOP specifies a boundary at 0EEC0H, leaving the space from 0EEC0H-0F6C0H for your subroutine.

```
—:F1:BASIC MEMTOP (0EEC0H)
 ISIS-II BASIC-80
 22620 BYTES FREE
```

If you invoke BASIC-80 on a 64K Intellec System without specifying MEMTOP, it looks like this:

```
—:F1: BASIC
 ISIS-II BASIC-80
 24668 BYTES FREE
```

If you locate your program as high as possible in free memory, BASIC-80 can make the most economical use of its remaining workspaces for string constants, variables, and strings.

When you have determined the optimum starting address for your subroutine, you can LOCATE it there with this command. LOCATE converts the relocatable object code to absolute object code, according to the starting address given. An example of giving the starting address for your subroutine code is shown below.

```
LOCATE :F1:PLMSUB.OBJ TO :F1:PLMSUB.LD CODE (0EEC0H)
```

This example converts the relocatable object code in :F1:PLMSUB.OBJ to absolute object code in the output file :F1:PLMSUB.LD, and makes all addresses in the subroutine relative to 0EEC0H. Refer to the *ISIS-II User's Guide* for further details of using LOCATE.

After LOCATE has converted your relocatable code to absolute object code, you can load it into memory with the ISIS-II Monitor DEBUG command. When you enter DEBUG followed by the filename of the load module, the subroutine is loaded into memory at the address specified in the load module. The starting address displays on your terminal. You then enter G8, which returns you to ISIS-II with the subroutine loaded at the specified address. From here you can invoke BASIC-80 and call the subroutine.

NOTE

You must give the MEMTOP option to reserve memory each time BASIC-80 is invoked.

Calling Subroutines

After a subroutine is loaded into memory, you can call it from BASIC-80. First, invoke BASIC-80 as you normally do, and give the MEMTOP option as previously specified.

The first step in calling a subroutine is defining its address with the DEFUSR function. Up to 25 subroutines can be addressed in this way, with an integer in the range 0-24. The starting address of the subroutine is given in hexadecimal:

```
DEFUSR5 = 0EEC0H
```

Once BASIC-80 knows where USR5 is located, you can call it. When you call it, you must supply any needed variables. Since the protocol for passing parameters follows PL/M conventions, you can only directly pass 2-byte integer variables or 2-byte addresses. If you specify a floating-point variable or a string variable you must use the VARPTR function to pass the address of the desired variable. For example, to pass the addresses of two floating-point numbers (K1 and K2):

```
120 A = USR5 (VARPTR(K1), (VARPTR(K2))
```

BASIC-80 goes to the address where K1 is stored (VARPTR(K1) and the address where K2 is stored (VARPTR(K2)). Once it has found these two values, it passes them to the subroutine.

If BASIC-80 encounters new variables after executing line 120 above, the memory locations of K1 and K2 can change, causing errors. Be sure that all variables are defined before using the VARPTR instruction.

In the example shown, once the parameters are passed, the subroutine executes. Because of PL/M-80 calling conventions BASIC-80 expects the returned result to be a two-byte integer in the HL register pair, and assigns this 16-bit value to A.

To return a single-precision floating-point, double-precision floating-point, or string result, you must use the appropriate data type character (see table 2-5 for a list of these characters) before the subroutine number in the USR function. For example, to tell BASIC-80 that a user-written subroutine returns a double-precision floating- point value:

```
240 A# = USR #15 (VARPTR(L#), A1)
```

In this example, the sharp sign (#) following USR tells BASIC-80 to reserve an 8- byte space for the double-precision result of USR #15. A# is also defined as double- precision, but the parameters passed may be of any numeric type.

Your subroutine must interpret the *first* parameter passed to it as the storage address of A#, and it must also place the result of USR #15 there.

References to string parameters are handled in a similar manner. VARPTR of a string is the address of the string descriptor, not the string data. Thus if a user subroutine returns a string then the user should code USR$[n]. BASIC allocates a 255-character string and pass the address of the string descriptor to the subroutine. Your routine may change the string length in the string descriptor to return a shorter string, but may not change the string's address. Neither parameter strings nor parameter string descriptors should be changed.

Array variables are passed as parameters by referencing the first element of the array. BASIC-80 follows row-major order.

To code in 8080/8085 assembly language, you must know the Intel format for representing integer, single-precision floating-point, and double-precision floating point numbers. Figure E-1 shows these representations.

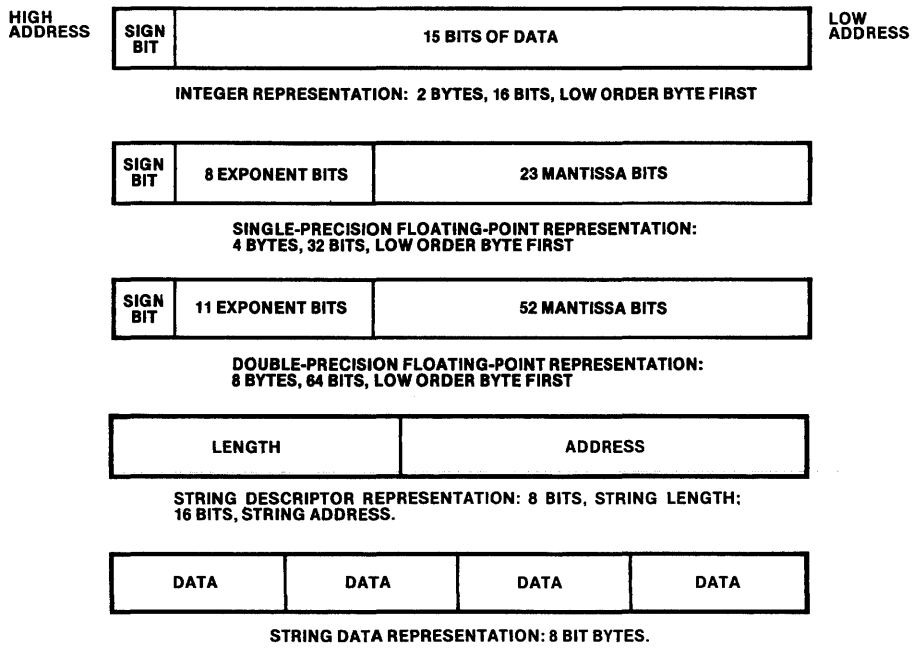

Figure E-1. Internal Representation of Numbers and Strings

Some Real Examples

The three sample programs provided in Figures E-2, E-3, and E-4 show how the same subroutine— adding three integer arguments—can be coded in FORTRAN-80, PL/M-80, and 8080/8085 assembly language. Notice that each program requires three parameters.

Once you have processed your subroutine through the PL/M-80 or FORTRAN-80 compilers, or through the 8080/8085 assembler, you can convert it to absolute object code and place it in memory with the LOCATE command. You must give the filename of your relocatable code file and the proper starting address for the subroutine code. In the example below, LOCATE returns absolute object code whose first byte is at address 0E000H:

```
—LOCATE :F1:PLMSUB.OBJ CODE (0E000H)
```

This command returns the absolute object file :F1:PLMSUB. You can now load this located code into Intellec memory at its proper address with the monitor DEBUG command. When you enter the DEBUG command with your filename, you invoke the Monitor, as shown in the example below. The Monitor responds with a period (.), expecting further commands:

```
—DEBUG:F1:PLMSUB
#0000
```

You can now return to ISIS-II and test your subroutine. Leave the Monitor and return to ISIS-II by entering G8 and a carriage return after the period:

```
.G8 (CR)
ISIS-II Vm.n
```

Invoke BASIC-80, and specify the highest memory address BASIC-80 can use with the MEMTOP option:

```
—:F1:BASIC MEMTOP(0DFFFH)
```

This prevents BASIC-80 from writing over your subroutine. BASIC-80 will appear and tell you how much free memory you have left:

```
ISIS-II BASIC-80 V1.0
18157 BYTES FREE
```

Once in BASIC-80 you can write programs that use your subroutine. The following BASIC-80 program defines your subroutine as USR1, asks for three integers, passes these three integer values to the subroutine, defines the result of the subroutine as the variable RESULT, and finally prints RESULT:

```
 10  REM THIS PROGRAM ASKS FOR THREE
 20  REM INTEGERS, PASSES THEM TO USR1, AND
 30  REM RETURNS THE RESULT.
 40  DEFINT A-Z
 50  DEFUSR 1=0E000H
 60  PRINT"ENTER THREE NUMBERS:"
 70  INPUT A,B,C
 80  PRINT "A=",A, "B=",B, "C=",C
 90  RESULT=USR%1(VARPTR(A), VARPTR(B), VARPTR(C))
100  PRINT"A+B+C="; RESULT
110  END
```

```
ASM80.OV3 :F1:USRASM.ASM

ISIS-II 8080/8085 MACRO ASSEMBLER, V2.0        USRASM   PAGE    1


  LOC  OBJ          SEQ          SOURCE STATEMENT

                      1 ;  ASSEMBLER ROUTINE TO TEST BASIC USR
                      2 ;
                      3 ;   RESULT=ARGA+ARGB+ARGC
                      4 ;   IGNORE OVERFLOW
                      5 ;
                      6          NAME    USRASM
                      7          PUBLIC  USRASM
                      8          CSEG
  0000 E1             9 USRASM:  POP     H          ;RETURN ADDR
  0001 220000   D     10         SHLD    RETADR     ;SET ASIDE
  0004 210000         11         LXI     H,0        ;SET TOTAL TO 0
  0007 CD1C00   C     12         CALL    ADDDE      ;ADD ARGC TO TOTAL
  000A 50             13         MOV     D,B        ;MOVE ADDR OF ARGB TO DE
  000B 59             14         MOV     E,C
  000C CD1C00   C     15         CALL    ADDDE      ;ADD ARGB TO TOTAL
  000F D1             16         POP     D          ;ADDRESS OF ARGA
  0010 CD1C00   C     17         CALL    ADDDE      ;ADD ARGA TO TOTAL
  0013 EB             18         XCHG               ;TOTAL IN DE
  0014 E1             19         POP     H          ;ADDR OF RESULT
  0015 73             20         MOV     M,E        ;LOW BYTE OF RESULT
  0016 23             21         INX     H
  0017 72             22         MOV     M,D        ;HI BYTE OF RESULT
  0018 2A0000   D     23         LHLD    RETADR     ;RETURN ADDRESS
  0018 E9             24         PCHL               ;RETURN
                      25 ;
                      26 ;  ADD 2 BYTES ADDRESSED BY DE TO HL
                      27 ;       CHANGES A,D,E,H,L
  001C E5             28 ADDDE:  PUSH    H          ;SAVE TOTAL
  001D EB             29         XCHG               ;ADDR IN L
  001E 5E             30         MOV     E,M        ;LOW BYTE TO BE ADDED
  001F 23             31         INX     H
  0020 56             32         MOV     D,M        ;HI BYTE TO BE ADDED
  0021 E1             33         POP     H          ;OLD TOTAL
  0022 19             34         DAD     D          ;ADD TO TOTAL
  0023 C9             35         RET
                      36 ;  DATA AREA
                      37         DSEG
  0000 0000           38 RETADR: DW      0          ;SAVED RETURN ADDRESS
  0000          C     39         END     USRASM

PUBLIC SYMBOLS
USRASM C 0000

EXTERNAL SYMBOLS


USER SYMBOLS
ADDDE  C 001C    RETADR D 0000    USRASM C 0000

ASSEMBLY COMPLETE,  NO ERRORS
```

Figure E-2. 8080/8085 Assembly Language Program

```
PL/M-80 COMPILER

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE PLMMODULE
OBJECT MODULE PLACED IN :F1:USRPLM.OBJ
COMPILER INVOKED BY:  PLM80 :F1:USRPLM.PLM


   1          PLM$MODULE:
                DO;
   2    1           USRPLM: PROCEDURE(PRESULT,PARGA,PARGB,PARGC);
   3    2             DECLARE (PRESULT,PARGA,PARGB,PARGC) ADDRESS;
   4    2             DECLARE (RESULT BASED PRESULT,
                             ARGA BASED PARGA,
                             ARGB BASED PARGB,
                             ARGC BASED PARGC) ADDRESS;
   5    2             RESULT=ARGA+ARGB+ARGC;
   6    2             END USRPLM;
   7    1           END PLM$MODULE;


MODULE INFORMATION:

     CODE AREA SIZE     = 0032H     50D
     VARIABLE AREA SIZE = 0008H      8D
     MAXIMUM STACK SIZE = 0004H      4D
     11 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

Figure E-3. PL/M-80 Program

```
FORTRAN COMPILER

   1              FUNCTION IRTN(IARGA,IARGB,IARGC)
   2              IRTN=IARGA+IARGB+IARGC
   3              RETURN
   4              END


MODULE INFORMATION:

     CODE AREA SIZE     = 002DH     45D
     VARIABLE AREA SIZE = 0008H      8D
     MAXIMUM STACK SIZE = 0004H      4D
     4 LINES READ

     0 PROGRAM ERROR(S) IN PROGRAM UNIT IRTN

     0 TOTAL PROGRAM ERROR(S)
     END OF FORTRAN COMPILATION
```

Figure E-4. FORTRAN-80 Program

APPENDIX F
RMX/80 BASIC-80

This appendix describes the differences between the RMX/80 version of BASIC-80 and the ISIS-II version, and tells you the requirements and procedures for generating disk-based or ROM-based versions of RMX/80 BASIC-80. It is recommended that you refer to the *RMX/80 User's Guide* and *RMX/80 Installation Guide* for supplementary information.

What is RMX/80 BASIC-80?

The RMX/80 BASIC-80 Interpreter (iSBC 802) runs under the RMX/80 Real-Time Multi-Tasking Executive. With RMX/80 BASIC-80, you can easily use the powerful computational and input/output capabilities of the iSBC/80 Microcomputer System, and apply them to solving a wide range of application problems.

The iSBC 802 software package gives you RMX/80 BASIC-80 in two forms. First, BASIC-80 modules, coded to run as tasks under RMX/80, are supplied on both single- and double-density diskettes. You can edit these modules with the ISIS-II Text Editor or CREDIT, and combine them with RMX/80 and user application tasks. You use the Intellec Development System to create a version of RMX/80 BASIC-80 tailor-made for your iSBC-based microcomputer system.

This first ("configure your own") option requires you to have a copy of the RMX/80 software and be familiar with its use. The portions of this appendix that describe how to configure your own RMX/80 BASIC-80 system will therefore make frequent references to the following publications:

RMX/80 User's Guide, manual order no. 9800522
RMX/80 Installation Guide, manual order no. 9803087-01

Second, if you want an "instant-on" BASIC-80 system, and have no need for additional software routines, you can use the predefined RMX/80 BASIC-80 configuration. This configuration includes three parts:

- The executable object module RMXSYS, supplied on both the single and double density diskettes.
- Two boot strap PROMs that load RMXSYS into the iSBC memory.
- A cable that connects the iSBC 204 disk controller to the bulkhead connector on any Intellec disk drive chassis.

With this predefined version, you have a version of RMX/80 that appears on RESTART on your iSBC hardware configuration. You need only make the necessary hardware connections, and BASIC-80 is ready to run.

You should be aware of the major differences between RMX/80 based BASIC-80 and ISIS-II based BASIC-80. These are:

- Configuring the RMX/80 modules determines how much memory will be available to the BASIC-80 interpreter and the program source. The sample configuration provides 8.4K bytes of memory space.
- You must also specify the number of files available to BASIC-80 when you configure the RMX/80 modules. The sample configuration supports 6 open files at once.

- With RMX/80 BASIC-80, you can create BASIC programs and program them into PROMs for permanent reference.
- You can configure RMX/80 to execute a PROM-resident BASIC-80 program immediately on restart.
- If you wish to interrupt program execution when BASIC-80 expects input from the console, you must enter CONTROL-C followed by a carriage return. If you wish to interrupt a program that has been stopped by a CONTROL-S, type CONTROL-C followed by CONTROL-Q.
- RMX/80 BASIC-80 does not have the EDIT Mode features supported by ISIS-II BASIC-80.
- RMX/80 BASIC-80 supports only the console and disk files.

You should also be aware of two similarities between RMX/80 BASIC-80 and ISIS-II BASIC-80:

- The data formats and protocols of the ISIS-II and RMX/80 disk file systems (DFS) are identical. You can create data or program files on one system and use them on the other.
- If you use a random file or open a file for output on one disk, and then remove that disk and insert another, you may destroy files on the new disk. You can open files for input without problems.

Initializing the Predefined RMX/80 BASIC-80 Configuration

RMX/80 allows many different configurations of its modules. Intel supplies two PROMs that you can use to load BASIC-80 from disk and to execute it. If you have the required hardware, as listed below, you can quickly run the predefined RMX/80 BASIC-80 configuration.

- An iSBC 80/30 Single Board Computer
- An additional 32K bytes of RAM on two iSBC 016s, or one iSBC 032 board.
- An iSBC 204 Disk Controller.
- An MDS-2DS Disk Drive attached through the cable supplied with the BASIC-80 package.
- An RS-232C compatible terminal.
- An iSBC chassis and power supply

If you have the required hardware, it must be configured as shown below. Refer to the *RMX/80 Installation Guide* if you must alter your present configuration. Table F-1 shows the jumper configurations described in the following paragraphs.

1. The iSBC 80/30 board must be wired to conform to the RMX/80 terminal handler interrupt structure. This requires the wiring changes listed in Table F-1. Check your iSBC 80/30 board to verify that these changes have been made. For further information, refer to the *RMX/80 User's Guide*.
2. The iSBC on-board PROMs must be addressed at locations 0000-0FFF. This is the factory-wired default configuration. These default jumper settings are shown in Table F-1. For further information, refer to the *iSBC 80/30 Single Board Computer Hardware Reference Manual*, chapter 2.
3. The iSBC 80/30 on-board RAM must be addressed at locations 4000H-7FFFH. This is the factory-wired default configuration. These default jumper settings are shown in Table F-1. Jumper 180-171 is a factory-wired default configuration that disables off-board access of on-board RAM. For further information, refer to the *iSBC 80/30 Single Board Computer Hardware Reference Manual*, chapter 2.

Table F-1. Sample Configuration Jumper Wiring

Board	Connect Jumper	Remove Jumpers
iSBC 80/30 Interrupt Handling	131-152 (INTERRUPT 2) 141-132 (EVENT CLK-IR1) 47-51 (CLK 0-A12-11) 143-127 (RXR INTR-IR 6) 142-126 (TXR INTR-IR 7) 145-140 (Ground INTR 5.5) 145-139 (Ground INTR 6.5)	123-138 (COUNT OUT-INTR 7.5) 46-47 (CLK 1-CLK 0) 47-52 (CLK 0-CLK 2)
iSBC 80/30 PROM Addressing 0000H-0FFFH	112-113 157-158 100-101 104-103 155-156 86-85	
iSBC 80/30 RAM Addressing 4000H-7FFFH	98-92 180-171 W1 at position A-B	
iSBC 016 RAM Addressing 8000H-0FFFFH	Board 1 - 7-6 Board 2 - 7-5	
iSBC 204 Base Addressing	S2 Settings* 7 ON 6 OFF 5 OFF 4 OFF	

*These numbers refer to the silk-screened numbers on the PC board, not to the numbers on the switch bank.

4. If you are using two iSBC 016 Random Access Memory boards, you must jumper one to supply RAM memory from locations 8000H-0BFFFH and the other to supply RAM from locations 0C000H to 0FFFFH. Table F-1 lists these jumpers. On one board, jumper 7-6 must be connected, which enables memory at locations 8000H-0BFFFH. On the other board jumper 7-5 must be connected, enabling memory at addresses 0C000H-0FFFFH. Refer to the *iSBC 016 16K RAM Board Hardware Reference Manual* for further information.

5. The iSBC 204 Disk Controller must be set to base address 80H and set at interrupt level 2. The base address is set by switch S2. The four switch settings of S2 are shown in Table F-1. Interrupt level 2 is the factory-wired default configuration. To verify interrupt level 2 operation, check for a wire jumper connecting terminal posts 63 and 67. Table F-1 lists these connections. For further information, refer to the *iSBC 204 Flexible Diskette Controller Hardware Reference Manual*, chapter 2.

6. The iSBC 204 Disk Controller must be the highest priority bus master. You can place the iSBC 204 in the top chassis slot with the iSBC 80/30 directly under it, or you can rewire the chassis backplane appropriately (see the *iSBC 80/30 Hardware Reference Manual* for details). The supplied cable attaches from J1 of the iSBC 204 controller to the plug on the rear of the MDS-2DS disk drive.

When you have configured your system, follow these steps to initiate BASIC-80:

1. Insert the supplied PROM1 in socket 0 and PROM2 in socket 1 of the iSBC 80/30.
2. Turn on power to the disk drives and iSBC system.
3. Insert your single density BASIC-80 disk into drive 0.
4. Type an upper case U at the terminal keyboard until the sign-on notice prints:

   RMX/80 BASIC-80 Vm.n

Generating Boot-Loaded and PROM-Based Versions of RMX/80 BASIC-80

Intel supplies two diskettes with the following modules. One diskette is single density; the other double density. You can modify and configure these modules to suit many possible combinations of hardware and software.

BASIC-80 Source Files

BQOPS.ASM — This module contains options used by other assembler modules. Figure F-1 shows a sample listing of this module.

BOOTCM.ASM — This module configures the BASIC Boot Loader.

BQBMEM.ASM — This module allocates memory for the BASIC Boot Loader.

BASCM.ASM — This module configures BASIC.

BQMEM.ASM — This module allocates memory for BASIC.

BASIC-80 Object Files

BASIC.LIB — This library contains all modules used by BASIC except for the following, which are generated by assembling the corresponding .ASM modules:

BOOTCM.OBJ
BQBMEM.OBJ
BASCM.OBJ
BQMEM.OBJ

CLOCK.OBJ — This module is a dummy clock module used with the iSBC 80/10.

BASIC-80 Executable Files

BQBOOT — This module is the Boot Loaded system, which is also provided on PROM.

RMXSYS — This module is the version of RMX/80 BASIC-80 which is loaded by the Boot Loader.

Software Requirements for Generating RMX/80 BASIC-80

You must have the following software tools and modules available in the appropriate drives to generate versions of BASIC-80.

Drive 0: — In drive 0, you must have a disk with these modules:

ISIS-II V3.4 (or later version)
All of the modules described above.
Link, Locate, and the ASM80 Macro Assembler

Drive 1: In drive 1, you must have a disk with these modules:

The RMX/80 nucleus, factory-configured for an iSBC 80/10, 80/20, or 80/30.

The RMX/80 extension files, including the Disk File System, and the RMX/80 Boot Loader files for the appropriate CPU board.

Once these two disks are present in the proper drives, you can begin configuring your RMX/80 BASIC-80 version. We'll look at procedures and examples of two kinds of BASIC-80s—a boot-loaded version and a PROM-based version.

Generating a Boot-Loaded RMX/80 BASIC-80

The software requirements for generating a boot loader and a compatible version of BASIC-80 are listed above. You must have the two disks in their proper drives with the given modules on each disk.

The hardware requirements for the boot loader and its accompanying version of BASIC-80 are:

An iSBC 201, iSBC 202, iSBC 204, or iSBC 206 Disk Controller, with its necessary cables, and the disk drives it controls.

An iSBC 80/10, iSBC 80/10A, iSBC 80/20, iSBC 80/20-4, or iSBC 80/30.

At least 48K bytes of RAM if DFS is used, or 32K bytes of RAM if it is not used. If you use an iSBC 80/30, the 16K bytes of RAM on-board count toward the total. You can use any combination of iSBC RAM boards.

An RS-232C compatible terminal, and accompanying cables.

There are three steps to generating a boot loader for RMX/80 BASIC-80:

1. You must examine and, if necessary, modify the BQOPS.ASM module.

2. You must also examine the BOOTCM and BQBMEM modules and modify them as needed.

3. After you have verified that the various assembly modules accurately reflect your hardware and software configurations, you SUBMIT the GBOOT.CSD module, shown in figure F-2, which assembles BOOTCM.ASM and BQBMEM.ASM and LINKs and LOCATEs the boot loader.

Modifying the BQOPS.ASM Module. The BQOPS.ASM module contains data used by the BOOTCM.ASM, BQBMEM.ASM, BASCM.ASM, and BQMEM. ASM modules. In most cases, this will be the only module you will modify before assembling modules for a new configuration. The boot loader configuration requirements are dependent upon the CPU model, the disk controller model, the number of disk drives available, and the highest memory location; if these do not change from configuration to configuration, then the boot loader need not be regenerated.

With the ISIS-II Text Editor or CREDIT, you can edit the BQOPS.ASM module listed in figure F-1 to support your exact configuration. Refer to the *ISIS-II User's Guide* for an explanation of the Editor's features and capabilities.

```
CPU      SET      30  ; MODEL OF CPU
BOOTED   SET       1  ; 1 IF BOOT VERSION, ELSE 0
TERMH    SET       1  ; 1 FOR FULL TH,0 FOR MINI
RATE     SET       0  ; BAUD RATE FACTOR
CONTR    SET     204  ; CONTROLLER NUMBER
DFS      SET       6  ; NUMBER OF DFS FILES USED
UIO      SET       0  ; 1 IF USER I/O DRIVERS ELSE 0
NFILES   SET       6  ; TOTAL FILES
HIRAM    SET  0FFFFH  ; HIGHEST RAM LOCATION
BOTMEM   SET  0FD40H  ; BOTTOM OF BOOT LOADER RAM
```

Figure F-1. Sample Configuration BQOPS.ASM Module

Each of the options in the sample configuration BQOPS.ASM module listed above is explained in the following paragraphs.

CPU — This option specifies the type of CPU used: 10 for iSBC 80/10 or 80/10A, 20 for iSBC 80/20 or 80/20-4, or 30 for iSBC 80/30. It is only referenced by the BQMEM.ASM module to initiate interrupt polling for iSBC 80/10 based DFS systems that are not boot loaded.

BOOTED — This option is used to allocate memory. It is 1 if the boot loader is used, or 0 if a PROM-based BASIC-80 is generated.

TERMH — With this option, a 1 specifies the Full Terminal Handler, and a 0 specifies the Mini Terminal Handler. The Mini Terminal Handler requires less RAM and PROM space.

RATE — This option generates an RQRATE to specify a baud rate if any non-zero value is given. For further information about setting baud rates, refer to the *RMX/80 User's Guide*.

CONTR — This option specifies the type of disk controller used. 201 indicates an iSBC 201; 202 indicates an iSBC 202; 204 indicates an iSBC 204; 206 indicates an iSBC 206.

DFS — This option specifies the number of DFS files you wish to have open at the same time. Specifying 0 means that DFS is not used.

UIO — This option enables your user-written I/O drivers if you specify 1. See "Adding User-Written I/O Drivers" in this Appendix for further details.

NFILES — This option specifies the combined number of DFS and user files that may be open at once. The number must be greater than or equal to the number specified in the DFS option.

HIRAM — This option specifies the highest RAM location available in the hardware configuration.

BOTMEM — This option places boot loader RAM at the highest possible location. This address should be 2BFH less than the address given in HIRAM.

After you modify BQOPS.ASM and verify the contents of the BOOTCM.ASM and BQBMEM.ASM modules, you are ready to generate your boot loader. To do this, you must assemble BOOTCM.ASM and BQBMEM.ASM, and LINK and LOCATE the resultant object code. The GBOOT.CSD module will do this with the SUBMIT command. Figure F-2 is a listing of the GBOOT.CSD module used with the BQOPS.ASM module in figure F-1.

The DATA location and the BOTMEM address must be the same. If your hardware configuration uses an iSBC 80/20 or iSBC 80/10, each occurrence of 830 should be changed to 820 or 810. After you have generated the boot loader , it should be burned into PROM and inserted into your CPU board. (See the *Universal PROM Programming Manual* for details)

The iSBC 80/10 does not have an onboard clock. If your configuration includes a clock, add the appropriate routines when linking GBOOT.CSD. (Refer to Appendix G of the *RMX/80 User's Guide* for further information.) If you don't have a clock in your configuration, include the dummy clockroutine CLOCK.OBJ.

```
ASM80   :F0:BOOTCM.ASM MACROFILE(:F0:)  NOSYMBOLS
ASM80   :F0:BQBMEM.ASM MACROFILE(:F0:)  NOSYMBOLS
LINK    :F1:BOT830.LIB(VECRST)
        :F1:RMX830.LIB(START)
        :F0:BOOTCM.OBJ,  &
        :F0:BQBMEM.OBJ,  &
        :F1:BOT830.LIB,  &
        :F1:DIO830.LIB,  &
        :F1:DFSUNR.LIB,  &
        :F1:RMX830.LIB,  &
        :F1:BOTUNR.LIB,  &
        :F0:PLM80.LIB    TO :F1:BQBOOT.LNK MAP PRINT(:F1:LNK.LST)
LOCATE  :F1:BQBOOT.LNK   TO :F0:BQBOOT MAP PUBLICS PRINT(:F1:LOC.LST)&
        CODE(40H)  DATA(0FD40H)  STACKSIZE(0)
```

Figure F-2. Sample Configuration GBOOT.CSD Module

Generating a Boot-Loadable BASIC-80. Once you have determined how to generate the boot loader that fits your particular RMX/80 implementation, the bulk of your work is over. Generating BASIC-80 is relatively simple.

There are four steps to generating a boot loadable RMX/80 BASIC-80: assembling the BASCM.ASM and BQMEM.ASM modules, and linking and locating the resulting BASIC-80 into RMXSYS. The GBASIC.CSD module is a SUBMIT file that performs these steps. Figure F-3 shows a listing of the GBASIC.CSD module used with the sample configuration. If you are using an iSBC 80/10 or iSBC 80/20 based system, you need to modify the "830" references in the module to "810" or "820" with the ISIS-II Text Editor. For further information about using the Text Editor, refer to the *ISIS-II User's Guide*.

The CODE and START addresses should reflect the addresses at the start of system RAM. The following list shows typical starting addresses:

iSBC 80/10 or 80/10-A	3C00H
iSBC 80/20	3800H
iSBC 80/20-4	3000H
iSBC 80/30	4000H

NOTE

The RMX/80 Nucleus declares all interrupt exchanges except RQL1EX (used for the system clock) as EXTERNAL. This is because user interrupt tasks must define the exchanges as needed. Any of these interrupt exchanges not used in a system, and therefore not declared in user code, is treated as an unresolved external reference by the linker, the locater, ICE-80, and ICE-85. Messages issued by these products that refer to unused interrupt exchanges can be considered as warnings and ignored. The messages issued by the various products are:

Linker:	UNRESOLVED EXTERNAL NAMES xxxxx xxxxx etc.
Locater:	UNRESOLVED EXTERNAL REFERENCE AT xxxxH (two messages for each exchange)
ICE-80	ERR=069
ICE-85	*WARNING UNSATISFIED EXTERNALS

You should check the messages to be certain that none of them refers to anything other than an unused interrupt exchange. Appendix J of the *RMX/80 Reference Manual* shows one way to "tie off" references to unused interrupt exchanges in a configuration module.

```
ASM80  :F0:BASCM.ASM MACROFILE(:F0:) NOSYMBOLS
ASM80  :F0:BQMEM.ASM MACROFILE(:F0:) NOSYMBOLS
LINK       &
       :F1:LOD830.LIB(LODINI),&
       :F0:BASCM.OBJ,&
       :F0:RMXBAS.LIB,&
       PUBLICS(:F0:BQBOOT),&
       :F1:LOD830.LIB,&
       :F1:DFSDIR.LIB(SEEK,DIRECTORY,ATTRIB,DELETE,RENAME),&
       :F1:DIO830.LIB,&
       :F1:DFSUNR.LIB,&
       :F1:THI830.LIB,&
       :F1:THO830.LIB,&
       :F1:RMX830.LIB,&
       :F1:UNRSLV.LIB,&
       :F1:PLM80.LIB,&
       :F0:BQMEM.OBJ TO :F1:BQBAS.LNK MAP PRINT (:F1:LNK.LST)
LOCATE  :F1:BQBAS.LNK TO :F0:RMXSYS MAP PUBLICS PRINT(:F1:LOC.LST)&
       CODE(4000H)  STACKSIZE(0)  START(4000H)  P URGE
```

Figure F-3. Sample Configuration GBASIC.CSD Module

Generating a PROM-Based RMX/80 BASIC-80

You can also configure RMX/80 BASIC-80 to reside in PROM. This requires 33K bytes of PROM, 2.8K bytes of RAM, and 400 bytes of RAM for each DFS file BASIC-80 will use. You should also set aside as much RAM as possible as workspace for BASIC-80.

The configuration explained below does not use DFS. Accordingly, the CONTR, DFS, and NFILES options shown in the BQOPS.ASM module are set to 0, and there are no references to the DFS libraries in the GBASIC.CSD module shown.

With this configuration, memory must be organized as follows (see the *iSBC 80/20 Hardware Reference Manual*):

PROM:	0 - 25K
iSBC 80/20 onboard RAM:	30K - 32K
iSBC 016 RAM:	32K - 48K

This configuration uses an iSBC 80/20. It's more difficult to configure an iSBC 80/10 for a PROM-based BASIC-80, because of memory allocation, and we'll look at how to do this after explaining the iSBC 80/20 configuration.

There are three steps to generating a PROM-based BASIC-80:

1. You must modify or create a version of BQOPS.ASM that supports the options you need for your configuration. Figure 4 shows the contents of a BQOPS.ASM PROM configuration using an iSBC 80/20. Examine each option available; if any need to be changed, this file (as well as any other ISIS-II file) can be edited with the ISIS-II Text Editor or CREDIT. Refer to the *ISIS-II User's Guide* for further information about using the Text Editor.

```
CPU     SET     20      ;MODEL OF CPU
BOOTED  SET     0       ;ONE IF BOOT VERSION, ELSE 0
TERMH   SET     0       ;1 FOR FULL TH, 0 FOR MINI
RATE    SET     28      ;BAUD RATE FACTOR
CONTR   SET     0       ;CONTROLLER NUMBER
DFS     SET     0       ;NUMBER OF DFS FILES OPEN AT ONCE
UIO     SET     0       ;1 IF USER I/O DRIVERS ELSE 0
NFILES  SET     0       ;TOTAL FILES
HIRAM   SET     0BFFFH  ;HIGHEST RAM LOCATION
BOTMEM  SET     0H      ;BOTTOM OF BOOT LOADER RAM
```

Figure F-4. BOQOPS.ASM Module for PROM-Based BASIC-80

2. You must also assemble BASCM.ASM, the BASIC-80 configuration module, and BQMEM.ASM, the memory allocation module. If these modules need to be changed to fit your configuration, you can edit them with the ISIS-II Text Editor.

3. You must then use the appropriate GBASIC.CSD file with SUBMIT, which will LINK and LOCATE the proper modules and their library references. Figure F-5 lists a GBASIC.CSD module that you could use with the PROM-based configuration specfied in figure F-4.

```
ASM80   :F0:BASCM.ASM MACROFILE (:F0:) NOLIST NOSYMBOLS
ASM80  :F0:BQMEM.ASM MACROFILE(:F0:)  NOLIST NOSYM BOLS
LINK        &
        :F1:RMX820.LIB(START),     &
        :F0:BASCM.OBJ,&
        :F0:RMXBAS.LIB,&
        :F1:MTI820.LIB,&
        :F1:MTO820.LIB,&
        :F1:RMX820.LIB,&
        :F1:DFSUNR.LIB,&
        :F1:UNRSLV.LIB,&
        :F1:PLM80.LIB,&
        :F0:BQMEM.OBJ TO :F1:BQBAS.LNK MAP PRINT(:F1:LNK.LST)
LOCATE :F1:BQBAS.LNK TO :F0:RMXSYS MAP PUBLICS PRINT(:F1:LOC.LST)
        CODE(0H)  STACKSIZE(0)  START(0H)  PURGE DATA(7800H)
```

Figure F-5. Sample GBASIC.CSD Module for PROM-Based RMX/80 BASIC-80

Configuring PROM-Based RMX/80 BASIC-80 With or Without DFS. If you do not need DFS facilities, PROM requirements are reduced by 7K bytes and RAM requirements are reduced by 1.6K bytes. You can configure without DFS by:

1. Setting DFS to 0 in BQOPS.ASM before assembling BASCM.ASM and BQMEM.ASM.
2. Excluding the DFS libraries from GBASIC.CSD, as in figure F-5. The DFS modules RQRNMX, RQDELX, RQOPNX, and RQATRX will be unresolved externals, but they present no difficulties.

Configuring a PROM-Based BASIC-80 For An iSBC 80/10-A Based System. In a typical iSBC 80/10-A configuration, the memory allocation would look like this (refer to the *iSBC 80/10 and iSBC 80/10A Hardware Reference Manual.*

On board PROM: 0K to 8K
On board RAM: 15K to 16K
iSBC 016 RAM: 16K to 32K
iSBC 464: 32K to 57K

System PROM is discontiguous, making linking and locating the configuration module more difficult. Follow these steps:

1. Edit the BQOPS.ASM module to specify the desired options. A sample iSBC 80/10 module is listed below.

```
CPU      SET      10  ;MODEL OF CPU
BOOTED   SET       0  ;1 IF BOOT VERSION, ELSE 0
TERMH    SET       0  ;1 FOR FULL TH, 0 FOR MINI
RATE     SET       7  ;BAUD RATE FACTOR
CONTR    SET     204  ;CONTROLLER NUMBER
DFS      SET       6  ;NUMBER OF DFS FILES OPEN AT ONCE
UIO      SET       0  ;1 IF USER I/O DRIVERS ELSE 0
NFILES   SET       6  ;TOTAL FILES
HIRAM    SET  07FFFH  ;HIGHEST RAM LOCATION
BOTMEM   SET      0H  ;BOTTOM OF BOOT LOADER RAM
```

Figure F-6. BQOPS.ASM Module for PROM-Based iSBC 80/10 BASIC-80

2. LINK RMX810.LIB (START), BASCM.OBJ, and as many DFS and terminal handler system modules as will fit on on-board PROM into one module. Don't worry about unresolved external references—these will be resolved in step 6.
3. LOCATE the module LINKed in step 2 with CODE (0) and DATA (3C00H). This specifies that the program starts in PROM at address 0, and that the data location will be at 3C00H, or 15K. This is the location of iSBC 80/10 on-board RAM. Record the STOP address of DATA. This will be used as the entry DATA address for the second module. This creates the first module.
4. Link all other modules together with PUBLICS.
5. LOCATE module at CODE (location of iSBC 464 PROM) and DATA (STOP address +1). This creates the second module.
6. LINK first executable module with PUBLICS (second module). This resolves external references.
7. Re-LOCATE your first module with the same command as in step 3.

Figure F-7 is a SUBMIT file that carries out all of the above steps.

```
ASM80  :F0:BASCM.ASM MACROFILE(:F0)  NOLIST SYMBOLS
ASM80  :F0:BQMEM.ASM MACROFILE(:F0:) NOLIST NOSYMBOLS
LINK       &
       :F1:RMX810.LIB(START),&
       :F0:BASCM.OBJ,&
       :F1:DFSDIR.LIB(DIRECTORY,RENAME),&
       :F1:MTI810.LIB,&
       :F1:MTO810.LIB,&
       :F1:RMX810.LIB,&
       :F0:CLOCK.OBJ,&
       :F1:UNRSLV.LIB,&
       :F1:PLM80.LIB    &
        TO  :F1:BQBAS.LNK MAP PRINT(:F1:LNK.LST)
LOCATE :F1:BQBAS.LNK TO :F0:BQ10P.ONE MAP PUBLICS PRINT(:F1:LOC.LST) &
       CODE(0H)  STACKSIZE(0)  START(0H)  DATA(3C00H)
LINK   :F0:RMXBAS.LIB(BQBAS,BQCONC),&
       PUBLICS(:F0:BQ10P.ONE),&
       :F0:RMXBAS.LIB,&
       :F1:DFSDIR.LIB(SEEK,ATTRIB,DELETE),&
       :F1:DIO810.LIB(DISKIO,HAN204,VIOHD4),&
       :F0:CLOCK.OBJ,&
       :F1:DFSUNR.LIB,&
       :F1:RMX810.LIB,&
       :F1:UNRSLV.LIB,&
       :F1:PLM80.LIB,&
       :F0:BQMEM.OBJ TO :F1:BQBAS.LNK MAP PRINT(:F1:LNK.LST)
LOCATE :F1:BQBAS.LNK TO (:F0:BQ10P.TWO MAP PUBLICS PRINT(:F1:LOCA.LST) &
       CODE(8000H)  STACKSIZE(0)  DATA(426AH)
LINK   :F0:BQ10P.ONE,PUBLICS(:F0:BQ10P.TWO) TO :F1:BQBAS.LNK &
       MAP PRINT(:F1:LNK.LST)
LOCATE :F1:BQBAS.LNK TO :F0:BQ10P.ONE MAP PRINT(:F1:LOCB.LST) PUBLICS &
       CODE(0)  STACKSIZE(0)  START(0)  DATA(3C00H)
```

Figure F-7. GBASIC.CSD SUBMIT Module for iSBC 80/10 PROM-Based BASIC-80

Configuring DFS on an iSBC 80/10

If you're using DFS with a PROM-Based BASIC-80 on the iSBC 80/10, you must add a line of code to the GBASIC.CSD module distributed on diskette. This code specifies use of the iSBC 201, 202, 204, or 206 disk controller.

For the iSBC 201, 202 or 206, add : :F1:DIO810.LIB(V10HD1),&
For the iSBC 204, add : :F1:DIO810.LIB(V10HD4),&
just before the line : :F1:DFSUNR.LIB,&
in the GBASIC.CSD module.

The BQMEM.ASM module contains code that polls the interrupt lines to find the interrupt from the disk drives. Be sure this polling is initiated (refer to the *RMX/80 User's Guide*).

iSBC 80/10 System Clock

The iSBC 80/10 does not have an on-board clock. You should include the dummy clock routine CLOCK.OBJ when configuring an iSBC 80/10 BASIC-80. This routine has two side effects:

1. With the full terminal handler, you may have to type a character before the RMX/80 BASIC-80 sign-on message prints.

2. There is no disk drive time out. If you reference a drive that doesn't have a disk in it, BASIC-80 will wait until one is inserted.

Your clock routines, or :F0:CLOCK.OBJ, should be added in the LINK command of GBOOT.CSD (if your BASIC is boot loaded), or in GBASIC.CSD.

Adding BASIC-80 to an Existing RMX/80 Configuration

This section assumes that a user has an existing RMX/80 configuration and wants to add BASIC-80 to it. the supplied assemblies configuration source and submit files may be used for reference.

Configuration Requirements

Tasks. BASIC-80 is called BQBAS, and the task that waits for control C is called BQCONC. BQBAS should be given a stack length of 64, a low priority such as 240, and a default exchange of BQEXCH. BQCONC should be given a stack length of 48, a priority higher than BQBAS (such as 200), and an initial exchange of RQWAKE. RQWAKE is the terminal handler exchange that receives a message when control C is typed at the console.

If a user-written I/O driver is to be used, it should be given an initial exchange of BQOPNX. The name, stack size, and priority may have whatever values are appropriate for the task.

Initial Exchanges

BQEXCH should be declared as a public exchange.

If a user-written I/O driver is to be included, BQOPNX should be declared as a public exchange.

Public Variables

An area of RAM as large as possible should be allocated for BASIC-80 work space. The user must supply two routines, BQSMEM and BQEMEM which return the address of the first byte of BASIC-80 work space in registers H and L and the last byte of BASIC-80 work space, respectively. The HL register requirement is consistent with PL/M-80 Address procedures.

Example:

```
            PUBLIC  BQSMEKM,BQEMEM
            CSEG
BQSMEM:     LXI     H,SMEM
            RET
BQEMEM:     LXI     H,EMEM
            RET
            DSEG
SMEM:       DS      5000
EMEM:       DS      1
            END
```

A public word variable called BQPRUN must be defined. If the value is non-zero, BASIC-80 will attempt to load and run the BASIC-80 source program at the address specified when BASIC is initiated.

Examples:

```
            PUBLIC  BQPRUN
BQPRUN:     DW      0 ; no automatic PRUN
```

or

```
            PUBLIC  BQPRUN
BQPRUN:     DW      PRUNIT
PRUNIT:     DB      "1 PRINT 'THIS WILL PRINT WHEN BASIC
                    IS STARTED' "
            DB      13, 10, 26 ; CR, LF, Control Z
```

Three byte public variables define the files available to BASIC. BQUIO should be zero if user-written I/O drivers are included, otherwise it should be one, BQDFS should be equal to zero if DFS is not included in the configuration. If DFS is included, BQDFS should be equal to the number of DFS files that may be open at once. BQNFIL should be the maximum number of DFS and user files that may be open at one time.

Example:

```
            PUBLIC  BQUIO,BQDFS,BQNFIL
BQUIO:      DB      0
BQDFS:      DB      6
BQNFIL:     DB      6
```

Linking and Locating

RMXBAS.LIB must be added to the list of files given to the LINK program. If a user-written I/O routine is to be used, its object module must also be included.

No special information is needed to locate object code for a system that includes BASIC-80.

Adding User-Written I/O Drivers to RMX/80 BASIC-80

You can add your own I/O drivers to any configuration of RMX/80 BASIC-80, so that BASIC-80 input and output statements employ user-defined I/O drivers. BASIC-80 treats these drivers as files with the device label :L1:. This is the proper syntax, as shown in opening a sequential disk file for output to the I/O driver file :L1:List:

```
10 OPEN "0", #1, ":L1:LIST"
```

The remainder of the file name may be anything conforming to the ISIS-II filename conventions. BASIC-80 will use the user I/O driver whenever an OPEN command is issued for a filename with a device type of :L1:. The open request message is sent to the BQOPNX exchange instead of to the DFS RQOPNX exchange. The messages sent to BQOPNX are exactly the same as messages sent to the DFS exchange RQOPNX. Consult the *RMX/80 User's Guide* for details. Therefore, a user-supplied task called BQUSER must wait at the BQOPNX exchange and must supply an exchange address when OPEN messages are received. This task or another task waits at this exchange and handles READ (for input files) or WRITE (for output files) and CLOSE requests. Figure F-8 is an example of a user-written I/O driver.

Adding BASIC-80 USR Routines to a Configuration

You can call 8080/8085 assembly language, FORTRAN-80, or PL/M-80 routines from BASIC-80 with the USR function (see Appendix E).

These routines can also reside in PROM. For ease of use, dedicate one or more PROMs and their sockets to this purpose. In this way, you can burn different subroutines as different needs arise without altering the addresses of the routines. Changing routines becomes as simple as changing PROMs.

Adding PROM-Based BASIC-80 Programs to a Configuration

You can also burn BASIC-80 programs into PROM with the ISIS-II BAPROM utility program. BAPROM converts BASIC-80 programs saved in ASCII format (with the SAVE "filename", A option) into relocatable object module format. You can save these modules from either ISIS-II or RMX/80 BASIC-80. They can then be linked, if needed, located, burned into PROM, and then run with the PRUN command.

If you wish to add USR routines or source files created by BAPROM to a given configuration, you can add the object modules to the LINK command; there is no automatic way, however, to communicate the starting addresses to BASIC-80. You must use the PUBLICS option of LOCATE and check the LOCATE PRINT file to find starting addresses.

You can also execute a BASIC-80 program immediately upon restart. To do this, you must change the constant BQPRUN in BASCM.ASM to the address of the BASIC-80 program stored in PROM. Here's an example:

1. Convert START.BAS into START.OBJ with the BAPROM program.
2. Add START.OBJ to the LINK command in the GBASIC.CSD module.
3. Change:

```
BQPRUN:   DW      0
```

   to :

```
          EXTRN   START
BQPRUN:   DW      START
```

   in : BASCM.ASM

```
;
;       USER TERMINAL HANDLER TO OUTPUT FOR FILE :L1:
;
  NAME          BQUSER
        PUBLIC      BQUSER
        EXTRN       BQOPNX,RQSEND,RQWAIT,RQOUTX
        CSEG
; WAIT FOR MESSAGE AT BQOPNX
; OPEN AND CLOSE ARE IGNORED
BQUSER:         LXI       B,BQOPNX      :EXCHANGE FOR USER OPEN
        LXI         D,0       ;WAIT FOREVER
        CALL         RQWAIT
        PUSH         H       ;MESSAGE ADDR
        LXI         D,4       ;OFFSET OF TYPE
        DAD         D
        MOV         A,M       ;MESSAGE TYPE
        LXI         D,5       ;STATUS IS AT OFFSET 9
        DAD         D
        CPI         14        ;CLOSE TYPE
        JZ          ZSTAT      ;ZERO STATUS AND QUIT
        CPI         15        ;OPEN STAT
        JNZ         NOTOPN    ;NOT AN OPEN REQUEST
        PUSH         H        ;SAVE STATUS ADDR
        LXI         D,6        ADD 6 TO GET LOCATION
        DAD         D         ;OF AFR EXCHANGE
        LXI         D,BQOPNX ;OPEN EXCHANGE ALSO USED FOR WRITE MESSAGES
        MOV         M,E      ;LOW BYTE AFR
        INX         H        ;
        MOV         M,D      ;HI BYTE AFR
        POP         H        ;RESTORE STATUS POINTER
        JMP         ZSTAT     ;AND ZERO STATUS
NOTOPN:
        CPI         12        ;WRITE TYPE
        JZ          WRITE
; BAD MESSAGE TYPE - RETURN ERROR STATUS
        MVI         M,18     ;STATUS-UNRECOGNIZED TYPE
        JMP         ERRRET   ;RETURN MESSAGE AND QUIT
; PASS MESSAGE ON TO TH
WRITE:        LXI         B,RQOUTX        ;TH OUTPUT EXCH
        POP         D        ;MESSAGE ADDR
        CALL         RQSEND;SEND MESSAGE
        JMP         BQUSER ;WAIT FOR MORE
;
; ZERO STATUS AND RETURN MESSAGE
ZSTAT:        XRA         A
        MOV         M,A      ;LOW BYTE OF STATUS
ERRRET:         INX          H
        MOV         M,A      ;HI BYTE OF STATUS
; RETURN MESSAGE
        DCX         H        ;BACK UP TO
        DCX         H        ;   RESPONSE EXCH
        MOV         B,M         ;HI BYTE OF RESP EXCH
        DCX         H
        MOV         C,M         ;LOW BYTE
        POP         D     ;MESSAGE ADDR
        CALL         RQSEND   RETURN MESSAGE
        JMP         BQUSER   ;WAIT FOR MORE
        END
```

Figure F-8. Sample User-Written I/O Driver Routine

Altering BASIC-80 Workspace

The BASIC-80 workspace stores the current BASIC-80 program, variables, constants, file buffers, strings. It should be as large as is practical.

Table F-2. Sample Configuration Memory Requirements

Module	PROM (bytes)	RAM (bytes)
RMXBAS.LIB	22287	1415
BOOTCM.OBJ	87	197
BASCM.OBJ	151	538
BQMEM.OBJ	18	Note 1
BQBMEM.OBJ		
Note 1: BQMEM.ASM allocates DFS memory areas and the BASIC-80 workspace. DFS requires 700 bytes, plus 400 bytes per DFS file. An additional 80 bytes are required for the controller stack on a non-boot loaded DFS system. On a boot loaded system, BQBMEM.ASM allocates controller stack area.		

The BQMEM.ASM module contains two labels: BQSMEM and BQEMEM. These labels correspond to the starting and ending addresses of the BASIC-80 workspace. The distributed code is written to make the greatest possible area of memory available as workspace:

- BQMEM.OBJ is the last module linked, so the starting address of the workspace is at the top of all data areas. BQSMEM uses this address.
- ASEG and ORG force the controller addressable areas (if DFS is specified) and boot loader code (in a boot loaded system) to the top of memory. A variable FREE addresses the last free byte below these. FREE is used by BQMEM. Note: the boot loader work area RQPOOL is re-used by BASIC-80.

If you wish to fix the BASIC-80 work area to a specific length or location, BQSMEM and BQEMEM must be modified accordingly. If you want to reserve free memory for BASIC-80 to POKE data into, you need to know the address loaded by BQMEM. This can be determined by examining the code of BQMEM in BQMEM.ASM. In the distributed version, this address is 0F123H. Accordingly, to reserve 1500 bytes of string space and 1K bytes to POKE into, the command

```
CLEAR  1500,0F123H  1024
```

should be given. If you give this command, the memory between 0ED24H and 0F123H will be unused and available to BASIC-80.

Burning a BASIC-80 Program Into PROM

To burn a BASIC-80 program into a programmable read-only memory (PROM), you must first convert the BASIC-80 program to Intel relocatable object file format. Included with BASIC-80 is a program that does this conversion.

These are the steps required to burn a BASIC-80 program into PROM:

1. Save the program on disk in ASCII format (the A option of the SAVE command). This can be done with either RMX/80 or ISIS-II BASIC-80.
2. Convert the ASCII program file to a relocatable object file with BAPROM.
3. Convert the file to absolute object file format with LOCATE.
4. Read the converted object file into PROM using the UPM READ command with the OBJECT option.
5. Burn the file into PROM with the PROGRAM command.

BAPROM is a program (and the name of the file that contains it) that runs under ISIS-II. It converts an ASCII file to Intel 8080/8085 relocatable object file format. It is not a compiler; it transforms ASCII data to a form that can be LOCATEd. It requires an Intellec or Intellec Series II microcomputer development system, at least 32K of RAM, at least one disk drive, and a terminal. To actually burn the PROM, you also need a Universal PROM Programmer (UPP) and the Universal PROM Mapper (UPM) program.

The format of the BAPROM command:

```
:Fn:BAPROM input file  TO output file
```

BAPROM is the name of the file that contains the BAPROM program. If it isn't on the disk in drive 0, include the drive number in the filename.

input file is the name of the file that contains the ASCII form of the BASIC-80 program to be burned into PROM.

output file is the name of the converted absolute object file. If you don't specify an output filename, it is given the same name as the input file with an extension of OBJ.

BAPROM does not modify the source in any way except to add a control Z at the end.

For example, assume you have written and tested a thermostat control program in BASIC-80 and saved it in ASCII format with the name HEATER.BAS. You wish to burn the program into PROM. If BAPROM and LOCATE are on drive 0, your program is on drive 1, and you wish to locate it at 0E000H then the session would go as follows:

```
-BAPROM :F1:HEATER.BAS
```

Because no output filename is specified, it is :F1:HEATER.OBJ. BAPROM displays its message, then displays the size of the input file:

```
ISIS-II BAPROM, Vm.n
SIZE = nnnnH BYTES
```

The size (nnnnH) is in hexadecimal.

```
-LOCATE :F1:HEATER.OBJ CODE(0E000H)
ISIS-II LOCATER Vm.n
```

To actually burn the resulting file (:F1:HEATER) into PROM, you need the Universal PROM Mapper program (UPM) and a Universal Prom Programmer (UPP). Supposing your program is approximately 2K and you wish to burn it into one 2716 PROM you must install the 2716 Personality Card into UPP, place a 2716 PROM into Socket 2 and then enter the following commands at the console:

```
-UPM
*TYPE 2716
*SOCKET = 2
*OFFSET
7500
*OFFSET = 9500H
READ into 0 object file :F1:HEATER
PROGRAM from 0E000H to 0E7FFH Start 0
```

From a more detailed description of UPM and UPP see: *Universal PROM Programmer Reference Manual*, 9800133F.

A discussion of how to select the proper offset can be found on pp. 36-37 of the *Universal PROM Mapper Operator's Manual*, 9800236A.

INDEX

REQUEST FOR READER'S COMMENTS

The Microcomputer Division Technical Publications Department attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS ...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN U.S.A.

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, CA 95051 (408) 987-8080

Printed in U.S.A.