RCA 1800

MICROPROCESSORS

Use of Basic 1 Compiler/Interpreter CDP18S834 V4 with the RCA MicroDisk Development System MS2000

MPM-834 Suggested Price $2.00

Use of Basic 1 Compiler/Interpreter CDP18S834 V4 with the RCA MicroDisk Development System MS2000

RCA Solid State | Somerville, NJ • Brussels • Paris • London
Hamburg • Sao Paulo • Hong Kong

Foreword

The Basic 1 Compiler/Interpreter CDP18S834V4 is like the CDP18S834 described in MPM-234 issued 4-80, but is supplied on a microdisk and is designed to run on the RCA MS2000 MicroDisk Development System which contains the RCA CDP18S845 MicroDisk Operating System (MicroDOS)

Implementation of BASIC1 for MicroDOS requires a change in the starting point of memory address user space from 1200 to 1600, and changes in the utility commands. These changes are shown on subsequent pages.

In this Manual, the BASIC1 language is described and detailed operation information for the compiler and interpreter is given. It is recommended that the user carefully read the User Manual for the CDP1802 Microprocessor, MPM-201 for a detailed description of its instruction set and architecture, the MPM-201C Supplement for the Instruction Set for the RCA CMOS Microprocessors CDP1804A, CDP1805A, and CDP1806A, and the User Manual for the RCA MS2000 MicroDisk Development System, MPM-241.

Contents

Foreword 3
1. Introduction 5
2. Elements of Basic 1 Compiler/Interpreter 6
  Programs and Statements 6
  Numbers 6
  Variables 7
  Expressions 7
  Functions 8
    MOD 8
    AND 8
    OR 8
    XOR 8
    MAX 8
    MIN 8
    SGN 8
    ABS 8
    HEX 8
    RND 8
    INP 8
    USR 8
3. Statement Types 10
  Comments and Declarations 10
    REM 10
    ! 10
  Assignment 10
    LET 10
  Control 10
    GOTO 10
    GOSUB 11
    RETURN 11
    END 11
  Conditional Statement 11
    IF 11
  Input/Output Statements 11
    INPUT 11
    PRINT 12
    OUTPUT 12
  Disk-Related Statements 12
    WFLN 12
    RFLN 13
    DOUT 13
    DIN 13
    CLOSE 13
    WEOF 13
    TIN 13
    TOUT 13
    NOUT 13
  System Control Statements 13
    NEW 13
    RUN 14
    LIST 14
    RDOS 14
4. Programming in Basic 1 15
  Immediate Execution and Program Modes 15
  Multiple Commands Per Line 16
  Disk Operations 16
  Special Keyboard Control Characters 17
  How the Compiler Uses the ! Statement 17
  Loops and Subroutines 18
  Programming Examples 18
    Arithmetic Drill Program 18
    Geometric Print Pattern Program 19
  Built-in Subroutines for the USR Function 20
  Error Messages and Programming Debugging 20
5. Running Basic 1 Compiler/Interpreter 22
  Loading and Starting the Interpreter 22
  Loading and Running the Compiler 22

Appendix A—Formal Definition of Basic 1 Compiler/Interpreter 24
Appendix B—Summary of Basic 1 Repertoire 25
Appendix C—Summary of Error Messages 28
Appendix D—Scratchpad Memory Allocations in the Interpreter 29
Appendix E—ASCII-Hex Table 31

1. Introduction

Basic 1 provides the most fundamental of those functions normally attributed to the high-level language called Basic. It is an excellent language for the beginning computer programmer. It is easily learned, and elementary application programs may be developed quickly. For the more experienced programmer, Basic 1 forms the core of a system whose facilities may be extended indefinitely by the addition of machine language subroutines limited only by the system memory.

The Basic 1 Compiler/Interpreter gives the user the option of (1) developing and running programs in Basic 1 directly, or (2) converting these programs to executable object code capable of running at a greater speed.

The interpreter allows the user to write programs in Basic 1 with line numbers for later execution or without line numbers for immediate execution. The disk-related statements incorporated in the interpreter allow the programmer to save programs on a floppy disk for later recall.

The compiler enables the programmer to take any stored program written in Basic 1 and translate it into assembly language, giving the user the flexibility of specifying where in memory the program, variables, and stack are to reside. The output of the compiler can be assembled by the MicroDOS macroassembler (ASM8) to produce the directly executable object code. Programs compiled and assembled run at speeds much greater than those run directly through the interpreter.

2. Elements of Basic 1 Compiler/Interpreter

In this chapter, the format of Basic 1 is outlined and the individual functions are described.

Programs and Statements

A program in Basic is an ordered list of numbered lines. Each line can hold a maximum of 70 characters and may contain one or more Basic statements. Lines may be entered in any order and line numbers may be skipped. Basic 1 arranges the lines in numerical sequence regardless of the order of entry. Line numbers can run from 1 to 32767.

Fig. 1 contains two examples of the same program. In Fig. 1(a), the program contains one Basic 1 statement per line and the lines are numbered consecutively starting at 1. In Fig. 1(b), the program contains multiple Basic 1 statements per line and the lines are numbered in increments of 10 starting at 10. The type of line numbering in Fig. 1(b) permits the addition of new or corrected statements into the program without renumbering all the lines.

Blanks written on a line have no significance to Basic 1. All spaces before the first non-numeric character are totally ignored. After that, however, blanks are preserved in the memory copy of the statement (i.e., each blank character occupies one byte). Judicious use of blanks within lines can often improve the readability of a Basic program. For information regarding the use of multiple statments in a line, refer to Chapter 4, **Programming in Basic 1.**

Numbers

Basic 1 accepts numbers that are either decimal or hexadecimal. A decimal number is a sequence of decimal digits (0-9) optionally preceded by a sign (+ or −). If no sign is present the number is assumed to be positive. All numbers are stored as 16-bit signed integers. Positive values may range from 0 to 32767 ($2^{15}-1$) and negative numbers may range from −1 to −32768 (2^{15}). It should be noted that −32768 cannot be input directly. However, an expression can be evaluated to give the value −32768.

A hexadecimal number is any sequence of hexadecimal digits (0-9, A-F) preceded by a pound sign (#). Hexadecimal numbers can range from #0000 to #FFFF. Fig. 2 gives some examples of valid and invalid number constants.

```
1 LET A=5
2 LET B=10
3 LET C=A*B+3
4 PRINT C
5 END
```

(a)

```
10 LET A=5:LET B=10
20 LET C=A*B+3:PRINT C
30 END
```

(b)

Fig. 1—Sample Basic 1 programs.

```
0
1
-1
-32767
#ABCD
```

(a) Valid

5.5	(integers only)
250000	(number out of range)
7,200	(commas do not appear in a number)
BB55	(hex constants must be preceded by a "#" as #BB55)

(b) Invalid

Fig.2—Valid and invalid number constants.

Variables

A variable is designated by any single capital letter (A-Z). Each possible variable is assigned a unique two-byte location in memory. The value of the variable is the contents of that location (i.e., a number in the range −32768 to 32767).

Expressions

An expression is a combination of one or more numbers, variables, or functions joined by operators and possibly grouped by pairs of parentheses. The permissible arithmetic operators are:

- \+ Addition
- − Subtraction
- * Multiplication
- / Division

Expressions also allow the indirection operator "@" to mean the byte from the following memory address. The indirection operator may be followed by any valid expression including one encompassing another indirection operator. Fig. 3 gives some examples of valid expressions. Fig. 4 gives some examples of expressions using the indirection operator.

Whenever Basic 1 encounters an expression within a statement during its execution, it evaluates the expression, combining the numbers and the values of the variables and functions using the indicated operators. Internal sub-expressions within parentheses are evaluated first. Usually, parentheses make clear the order in which operations are to be performed. However, if there is an ambiguity because parentheses are absent, Basic 1 gives precedence to multiplication and division over addition and subtraction. Arithmetic operators also take precedence over the indirection operator. In cases involving two operators of equal precedence, evaluation would proceed from left to right. An expression may be optionally preceded by a sign. In the example expressions given in Fig. 5, the operators are numbered to indicate the order in which they are performed.

When the result of a division is not an integer, Basic 1 will truncate the result. Examples of this division and truncation are given in Fig. 6.

During the evaluation of an expression, all intermediate values and the final values are truncated to the lowest 16 bits of the results. That is, expressions are evaluated modulo 2^{16} with the most significant bit being the sign bit. No attempt is made to discover arithmetic overflow conditions except that

```
D        451      #2ABC          A+1-C      (((C)))
B-6*Z         (X+Y)/(A+B)      -(-46/(-32767+(I*I)))
```

Fig. 3—Valid expressions.

```
@125    = The contents of memory location 007D (125 = hex 007D)
@#7D    = An equivalent expression

@@5     = The contents of memory location whose address is in location 0005
@5+1    = The contents of memory location 0006

(@5)+1  = One more than the contents of location 0005
1+@5    = An equivalent expression
```

Fig. 4—Expressions using the indirection operator "@".

```
A+B*C                    (A+B+C)/5
 2 1                       1 2 3

A+B*6/4+                 (((A+B)*4)-6*D)
 3 1 2 4                      1  2 4 3

A+B-C+D                  A*B/C*D/E*F
 1 2 3                    1 2 3 4 5
```

Fig. 5—Expression evaluation. The numbers indicate the order in which the operations are performed.

DIVISION	RESULT
6/3	2
(−6)/(−3)	2
7/3	2
(−7)/(−3)	2
8/3	2
(−6)/3	−2
6/(−3)	−2
6/7	0

Fig. 6—Expressions using the division operator.

an attempt to divide by zero results in an error stop. Any expression which evaluates into the range of 32768 to 65535 (2^{15} to 2^{16}-1) has a sign bit of 1 (making it negative). Thus, it is actually treated by Basic 1 as if 65536 (2^{16}) were subtracted from it. As a consequence, the following are expressions which have the same value:

−4096
15*4096
2*16384/8
30720+30720

Functions

Function references may appear in expressions in the same way as a variable or number constant. They may also appear by themselves anywhere an expression is permitted. Following is a listing of the Basic 1 functions and a description of what each function does.

MOD Function

Form—MOD (Expression 1,Expression 2)

The MOD function divides the value of expression 1 by the value of expression 2 and returns the remainder. It is exactly equivalent to the expression

$$A-\left(\frac{A}{B}\right)B$$

When the OR, XOR, or AND function is used in the compiler, if the second argument is a two-byte hex constant with #00 in the high or low byte, insufficient code is produced to form an OR, XOR, or AND with the #00 byte of the argument. The user can correct the problem by editing the output from the compiler with the insertion of additional code to perform the missing OR-ing function. In the following example the code that the user should add is underlined.

```
OR (Y,#1500) should generate
  SEP L;DC A.O(VY)
  GHI ZZ;PLO AC
  GHI AC;ORI A.O(015H);PHI AC;
    GLO AC;ORI A.O(00H);PLO AC
  SEP 8;DC A.(VY)
```

AND Function

Form—AND (Expression 1,Expression 2)

The AND function returns the bit-for-bit logical AND of the two arguments.

OR Function

Form—OR (Expression 1,Expression 2)

The OR function returns the bit-for-bit logical OR of the two arguments.

XOR Function

Form—XOR (Expression 1,Expression 2)

The XOR function returns the bit-for-bit logical XOR of the two arguments.

MAX Function

Form—MAX (Expression List)

The MAX function returns the greatest value in the expression list. The expression list is a list of expressions separated by commas.

MIN Function

Form—MIN (Expression List)

The MIN function returns the smallest value in the expression list.

SGN Function

Form—SGN (Expression)

The SGN function returns a −1, 0, or +1 depending on whether the argument is negative, zero, or positive, respectively.

ABS Function

Form—ABS (Expression)

The ABS function returns the absolute value of its argument.

HEX Function

Form—HEX (Number,Width)

The HEX function is valid only in a PRINT statement. It prints the value of the number in hexadecimal. Both the number and the width can be any valid expression. If the width evaluates to a number between 1 and 3, it specifies how many digits (from the least significant part of the number) will be printed. Otherwise, or if the width parameter is omitted, four digits will be printed.

RND Function

Form—RND (Expression 1,Expression 2)

The RND function returns a positive random number in the range from the value of expression 1 to the value of expression 2, inclusive. The value of expression 2 must be greater than the value of expression 1. If the arguments are invalid, an error stop may occur.

INP Function

Form—Line Number INP (Port)

The INP function is used to input data from a specific port. It generates a hardware in instruction with the N lines set according to the parameter (Port). It is essential that the Port expression evaluate to a number in the range of 1 to 7. The INP function executes by returning the value of the data on the specified input port as a number in the range 0 to 255.

USR Function

Form—USR (Expression 1,Expression 2,Expression 3)

The USR function is an important feature of Basic 1 that allows the user to extend the features of the

language by means of machine language routines. To use this feature, the programmer must be familiar with machine language programming, the instruction set for the CPU (See MPM-201, User Manual for the CDP1802 COSMAC Microprocessor), MPM-201C, Instruction Set for the RCA CMOS Microprocessors CDP1804A, CDP1805A, CDP1806A, and the CPU registers that are available and can interface between the machine language program and the Basic 1 program. Because implementation of Basic 1 for MicroDOS requires a change in the starting point of memory address user space from 1200 to 1600, the programmer should also be aware of the areas in memory that can be used.

When the USR function is encountered in a program, Basic 1 evaluates the first expression and transfers control to that address. The second and third expressions are optional. If a second expression is included, it is evaluated and the resulting value is passed to the called program as the contents of CPU register 8. If a third expression is included, its value is passed in register A (with D also holding RA.0). The subroutine receives control with P = 3 and X = 2.

The called program must return with a SEP 5 (D5) instruction. When it returns, its 16-bit function value is the final contents of RA.1 and D (lower 8 bits in D) just before the SEP 5 was executed.

Machine language subroutines have the free use of R0, R1, R8, RA, and RF. In addition, R2 is pointing at a free byte in the control stack.

Basic 1 has a built-in call and return subroutine that preserves the accumulator (D) and destroys the low part of register RE. This subroutine differs from the Standard Call and Return Technique (SCRT) described in the User Manual for the CDP1802 COSMAC Microprocessor, MPM-201.

The user program area is located directly above the interpreter, starts at #1600, and runs up. The stack starts at the top of memory (#7FFF in a 32-kilobyte system) and runs down. See Fig. 7(a). The machine language subroutine can be positioned somewhere between the user program area and the stack, or the position of the user program area can be changed, thereby saving X number of bytes for the machine language subroutines. To change the position of the user program area, the constant #1600 stored at location #011C should be changed to some other value (#1800, for example). See Fig. 7(b).

Procedure: After the interpreter is loaded, press RESET and RUN U on the front panel of the development system. When the asterisk "*" prompt is returned, type in the command "I011C 1600 (CR)", and then execute a cold start, " P100". For more information on cold-starting the interpreter refer to Chapter 5, Running Basic 1. The block of memory between #1600 and #1800 will now be skipped by the interpreter.

The compiler permits the user to place the variables, program, and the stack anywhere in memory. The user must keep in mind, however, that bytes at locations #0000, #0001, #0002, and #0003 are always occupied. With this flexibility any amount of space that might be needed for machine language subroutines can be saved.

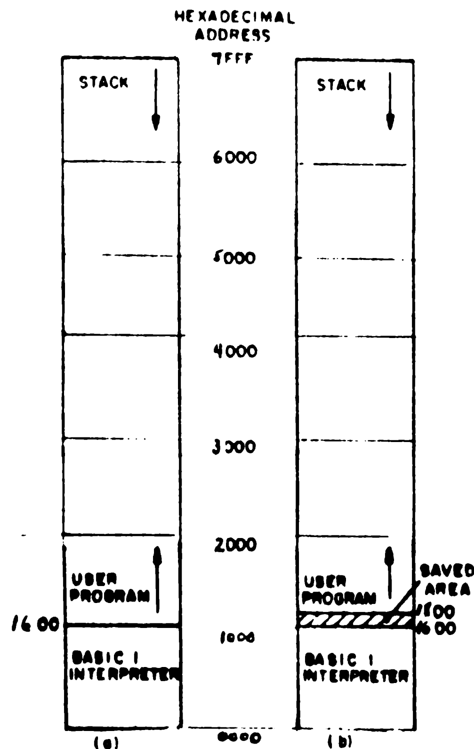

Fig. 7—Interpreter memory usage in 32-kilobyte system. (a) Unmodified. (b) Modified to save #0200 bytes for machine language subroutines.

3. Statement Types

A statement normally begins with a key word, such as PRINT or GOTO, indicating the type of statement. The interpretation of the rest of the statement depends on the key word. In some cases a shortened form of the key word is also acceptable. An example is PR instead of PRINT. In this chapter, the various types of statements and the associated key words will be described.

Comments and Declarations

REM Statement

Form—Line Number REM Text

The REM statement is used to insert remarks or comments into a program. During program execution, it is ignored by Basic 1.

Example—100 REM THIS IS A REMARK

! Statement

Form—Line Number ! Assembly Language

The ! statement is used to pass assembly language through the compiler. The interpreter treats the ! statement the same as the REM statement.

Example—100 !,T'THIS WILL BE PASSED THROUGH THE COMPILER'

The use of the ! statement will be explained in detail in Chapter 4, **Programming in Basic 1**.

Assignment

LET Statement

Form—Line Number LET Variable = Expression

The LET statement is used to assign a value to a variable. The value may be any general expression. The key word LET is optional.

Examples—100 LET A = 5
150 B = 10

The LET statement can be combined with the indirection operator "@" to store a byte anywhere in memory.

Form—Line Number LET@ Address = Datum

Both the address and the datum may be any valid expression.

Expression—100 LET@ #3000 = #2A

Control

GOTO Statement

Form—Line Number GOTO Expression

The GOTO statement transfers control in the program to the line number specified by the expression. If the expression contains one or more variables it is considered a computed transfer of control.

Examples—100 GOTO 150
150 GOTO 1*2 + 200

The interpreter is able to execute a program with computed transfers of control. For the compiler to work with such a program, however, it needs a table of target lines following the END statement of the form:

LINE NUMBER, TARGET LINE 1, ----, TARGET LINE 5, -1

LINE NUMBER, TARGET LINE 6, TARGET LINE 7, 0

where the -1 and the 0 at the end of the lines refer to table incomplete and table complete, respectively. Lines containing the table of target line numbers must have a line number of 32001 or greater.

Example—.

```
.
.
150 GOTO X*10 + 200 (where X can
                     range from 0
                     to 5)
.
.
.
300 GOTO Y*5 + 500 (where Y can
```

```
                        range from 0
                        to 3)
        .
        .
        .
        5000 END
        32001,200,210,220,230,-1
        32002,240,250,500,505,-1
        32003,510,515,0
```

GOSUB Statement

Form—Line Number GOSUB Expression

The GOSUB statement is used to call an internal subroutine. It is executed exactly the same as a GOTO statement with one exception. The number of the line immediately following the GOSUB statement is recorded by Basic 1.

Examples—100 GOSUB 1010
100 GOSUB 1*10+500

The GOSUB statement can be combined with the indirection operator "@" to jump to a machine language routine.

Form—Line Number GOSUB@ Expression 1,Expression 2,Expression 3

Expression 1 specifies the address of the machine language subroutine. Expressions 2 and 3 are optional. If expression 2 is included, it is evaluated and the resulting value passed to the called program as the contents of CPU register 8. If a third expression is included, its value is passed in register A (with D also holding RA.0). The subroutine receives control with P=3 and X=2, and must return with a SEP 5 (D5) instruction. (See USR Function in Chapter 2, **Elements of Basic 1 Compiler/Interpreter**.)

Examples—100 GOSUB@#3000
100 GOSUB@ 1300,A*B
100 GOSUB @ #4100,0,#56

RETURN Statement

Form—Line Number RETURN

The RETURN statement is used to mark the end of an internal subroutine. It executes by transferring control back to the statement whose line number was recorded as the result of the execution of a GOSUB statement. The acceptable short form of the key word RETURN is RET.

Examples—100 RETURN
100 RET

Internal subroutines are explained in Chapter 4, **Programming in Basic 1**.

END Statement

Form—Line Number END

The interpreter uses the END statement to terminate execution of a program and return to the enter mode. It must be the last statement executed in a program and there may be as many END statements in a program as needed. The interpreter accepts as a synonym for END the key word STOP.

Examples—100 END
100 STOP

The compiler uses the END statement to terminate compilation. In the compiler, the END statement must be the last line of the program. This statement must be the one and only END statement in the program. The key word STOP is not recognized by the compiler.

Conditional Statement

IF Statement

Form—Line Number IF Expression 1 Relation Expression 2 THEN Statement

The IF statement is used to compare two expressions according to the specified relation. If the condition specified is true, then the associated statement is executed. Otherwise, the program is advanced to the next line number. The permissible relational operators are:

=	equal
<	less than
>	greater than
<=	less than or equal (not greater)
>=	greater than or equal (not less)
<> or ><	not equal (greater than or less than)

The associated statement may be any other valid Basic 1 statement including another IF statement. The key word THEN may be omitted.

Examples—100 IF I>25 THEN END
100 IF I<5 PR "UNDER"
100 IF A>B IF B>C I=I+1
(Increments only if B is between C and A)

The IF statement also allows the key word GOTO to be replaced by THEN, thus making the two statements below equivalent.

Examples—100 IF B=5 GOTO 200
100 IF B=5 THEN 200

It should be noted that the key word THEN is essential here and cannot be omitted.

Input/Output Statements

INPUT Statement

Form—Line Number INPUT Inputlist

The INPUT statement is used to input data from the terminal or disk. The Inputlist is a succession of one or more variables separated by commas. The execution of this statement begins with the typing of a question mark prompt indicating that Basic 1 is ex-

pecting the user to type in data. The user should respond by typing a line of one or more expressions separated by commas and terminated with a carriage return. Each input expression is evaluated and assigned to its input variable in the INPUT statement.

If the number of requested variables in the Inputlist is not satisfied by the number of expressions in the user's input line, a new ? prompt will be issued asking for more information. If the number of expressions in the user's line is greater than the number of requested variables, then those input expressions not requested are saved internally.

The key word INPUT can be shortened to IN. This short form has a slight functional difference. The INPUT statement will unconditionally read another line, ignoring any remaining data in the current line. The IN statement, however, will check to see if any saved expressions exist. If so, then these saved expressions are used first to satisfy some or all of the variables requesting values. Only when no saved data exists is the ? prompt issued.

Examples—100 INPUT A,B,C
100 IN A,B,C

The key words INPUT and IN can be followed by a prompt string. The string is a group of characters enclosed in quotes that will be typed in front of the first ? prompt when the statement is executed. If the prompt string is followed by a comma, a new line of input is required. If there is no comma, the existing line of data is used until exhausted. The prompt string over rides key word spelling.

Examples—100 INPUT "READ", A,B,C
100 IN "GIVE IT TO ME" A,B,C

PRINT Statement

Form—Line Number PRINT Printlist

The PRINT statement is used to print data or messages to the console printer or disk. The Printlist is a succession of one or more items to be printed separated by either commas or semicolons. Each print item may be either an expression or a character string enclosed in quotes. In the first case, the value of the expression is typed. In the second case, the character string is printed verbatim. No spaces are generated between the printout of items separated by semicolons in the PRINT statement. On the other hand, the printout of an item preceded by a comma in the PRINT statement begins at the next tab setting. Tabs are automatically set every eight characters. Commas and semicolons, character strings, and expressions may be mixed in one PRINT statement in any manner. The acceptable short form for PRINT is PR.

Examples—100 PRINT 1,2;3,"HELP" prints as:

```
1        23        HELP
```

100 PR 5;6;7,8,9 prints as:

```
567      8         9
```

Normally, the execution of a PRINT statement terminates with the generation of a carriage return and line feed to begin the new line. If the PRINT statement ends with a comma or semicolon, however, then the carriage return-line feed sequence is suppressed, permitting subsequent PRINT statements to output on the same line or permitting an input message to appear on the same line as the previous output.

The interpreter will accept the leading quote of a character in place of the key words PRINT and PR.

Examples—100 "HELLO"
prints as: HELLO
100 "HELLO",5;6,7
prints as: HELLO 56 7

The compiler will not accept this last form as a valid statement.

OUTPUT Statement

Form—Line Number OUTPUT Port,Data

The OUTPUT statement is used to generate a hardware out instruction with the N lines set according to the first parameter (Port), and the output data supplied in the second parameter (Data). Either parameter may be any valid expression, but it is essential that the Port expression evaluate to a number in the range 1 to 7 or no output may occur. Only the least significant 8 bits of the Data expression are output. The accepted short form of the key word OUTPUT is OUT.

Examples—100 OUTPUT 5,6
100 OUT 7,1

Disk-Related Statements

WFLN Statement

Form—Line Number WFLN

The WFLN statement is used to initialize an output file (write file).

Example—100 WFLN

This statement executes by first closing the last output file (if one existed) and setting the internal flags for terminal input and console printer output. Then, it prompts the user with a WRITE? for a FILENAME. The user must respond by typing in a FILENAME in the form:

(NAME).(EXTENSION):(DRIVE)

The NAME consists of from one to six alpha-numeric characters. The EXTENSION consists of from one to three alpha-numeric characters. The first character of each must be an alphabetic character. The DRIVE must be a number, either 0 or 1. Both the EXTENSION and the DRIVE are optional. If the DRIVE is not specified, a default value of 0 is used. The following are examples of valid FILENAMES:

TRIAL1.BSC:1
TRIAL1:1
TRIAL (DRIVE default value is zero)
T5529.A17 (DRIVE default value is zero)

If an invalid FILENAME is typed in, an error message will be printed and the user will be prompted with a ? for another FILENAME. Once a valid FILENAME has been received, a file is opened; that is, space is allocated on the disk under that FILENAME.

RFLN Statement

Form—Line Number RFLN

The RFLN statment is used to initialize a read file (input file).

Example—100 RFLN

The RFLN statement executes the same way as the WFLN statement but with a couple of minor changes. First, an input file is involved rather than an output file. Second, instead of a WRITE? prompt being issued, a READ? prompt is issued.

DOUT Statement

Form—Line Number DOUT Filename

The DOUT statement is used to set up a disk output file. This statement executes by closing the last output file (if one existed), processing the Filename found in the statement, opening an output file under the new Filename, and setting internal flags for disk output. The Filename is of the same form as described under the WFLN statement. If an invalid Filename is discovered, the interpreter will return to the enter mode and the compiler will return control to CDOS.

The DOUT statement can be used without the Filename. When so used, only the internal flags for disk output are set.

Examples—100 DOUT STORE1.TR1:1
100 DOUT

DIN Statement

Form—Line Number DIN Filename

The DIN statement is used to set up a disk input file. This statement executes the same way as the DOUT statement but with some minor changes. First, input files are involved instead of output files. Also, when internal flags are set, the setting will be for disk input.

Examples—100 DIN TRBLE:1
100 DIN

CLOSE Statement

Form—Line Number CLOSE

The CLOSE statement is used to close a disk output file and set the internal flags for console printer output.

Example—100 CLOSE

WEOF Statement

Form—Line Number WEOF

The WEOF statement is used to put an end-of-file character (DC3) in a file. The execution of this command also sets the internal flags for console printer output. It is used for separating lines of data in a disk file.

Example—100 WEOF

TIN Statement

Form—Line Number TIN

The TIN statement restores the terminal as the primary input device. Because this statement does not alter the disk control block, further use of the DIN statement without a parameter will resume where the previous reference left off. Any error stop or the execution of a NEW or END statement will force an implicit TIN.

Example—100 TIN

TOUT Statement

Form—Line Number TOUT

The TOUT statement restores the console printer as the primary output device. Because this statement does not alter the disk control block, further use of the DOUT statement without a parameter will resume where the previous reference left off. Any error stop or the execution of a NEW or END statement will force an implicit TOUT.

Example—100 TOUT

NOUT Statement

Form—Line Number NOUT

The NOUT statement disables the output. It is cancelled by a TOUT or DOUT statement.

Example—100 NOUT

System Control Statements

System control statements are recognized only by the interpreter. They are not normally included as part of a program. They are normally entered without a line number.

NEW Statement

Form—NEW

The NEW statement is used before a new program is entered. Execution of this statement clears the program area in memory. Alternate acceptable forms of the key word NEW are CLEAR and SCR.

Examples—NEW
CLEAR
SCR

RUN Statement

Form—RUN Expression Sequence

The RUN statement is used to start execution of a program. It begins execution at the first (lowest) line number. If the key word RUN is followed by a comma followed by a sequence of one or more expressions separated by commas, then the expression sequence is treated as an initial input line. This line will be scanned first when IN statements are executed.

Examples—RUN 1,2,3
RUN

LIST Statement

Form—LIST Expression 1,Expression 2

The LIST statement is used to print out all or part of a stored user program. Both parameters are optional. If no parameters are given, the whole program is listed. A single expression parameter is evaluated to a line number. If the line exists, it is printed. If both parameters are given, all lines with line numbers in the range specified are printed.

Examples—LIST
(Prints entire user program)
LIST 100
(Prints line 100 if it exists)
LIST 1,1000
(Prints all lines between 1 and 1000 inclusive)

RDOS Statement

Form—RDOS

The RDOS statement is used to restore control to the Disk Operating System.

Example—RDOS

4. Programming in Basic 1

This chapter introduces the user to programming in Basic 1. Ground rules for the interpreter and the compiler are discussed as well as some programming techniques. Two programming examples are given and error messages and programming debugging are covered.

Immediate Execution and Program Modes

After the user loads the interpreter, a colon prompt ":" is returned indicating that the interpreter is in the enter mode and is ready to accept a line. After each input line is handled, the interpreter returns to the enter mode. If the user inputs a line without a line number followed by a carriage return (CR), the interpreter goes into the **immediate execution mode**. In this mode, the line typed is executed immediately. If the user inputs a line preceded by a line number and followed by a (CR), that line would be taken from the input buffer and stored in the user program area. Typing the RUN command would then put the interpreter into the **program mode** and the lines stored in the user area would be executed.

One important use of the immediate execution mode is to permit line-at-a-time testing. LET, IF, and PRINT can be demonstrated this way.

The INPUT and IN statements can also be directly executed but, because of the way Basic 1 buffers its input lines, the INPUT or IN statement cannot be directly executed for more than one variable at a time. Thus, if the following statement is typed without a line number

```
INPUT A,B,C
```

only the value of A will be asked for. The values of B and C will remain as they were. If the statement

```
IN A,B,C
```

is typed without a line number, the value of B will be copied to A and only the value for C will be requested. Similarly, the statement

```
IN A,5,B,10,C,15
```

will execute directly (loading A, B, and C with the values 5, 10, and 15, respectively) and request no input. But, with a line number in a program, this statement will produce an error stop after requesting one value.

Clearly, there is no point to executing REM or END in the immediate mode. Furthermore, GOSUB and RETURN are normally meant for the program mode. On the other hand, an immediate GOTO has the same effect as if RUN were typed, but execution may begin at other than the program's first statement.

The program mode is entered to execute a program stored in the user area. As noted before, lines preceded by a line number are stored in the user area. No errors, therefore will be detected in a stored program until an attempt is made at execution.

System control statements are not normally included in a stored program. NEW (as well as CLEAR and SCR) obviously should be avoided because execution of any of these commands would result in a self-destruct. A stored RUN, however, will be the equivalent to a GOTO to the first statement. Also, a LIST statement may be included as part of a program and used for printing out large text strings, such as instructions to the operator.

It should be noted that the interpreter will accept system control statements in the body of a stored program but the compiler will not. Consequently, no system control statements can be present in a program intended to be compiled.

Multiple Commands Per Line

An important feature of Basic 1 is the ability to string statements together on a line. This feature is accomplished simply by separating the statements with a colon ":". An example is

```
100 PRINT "START":LET A=1:GOTO 300
```

Statements which effect a transfer of control cannot be followed by any other statement on the same line, but otherwise any combination of valid Basic 1 statements is acceptable. This capability is particularly useful when the result of some conditions requires two or more actions. For example,

```
150 IF X<0 PRINT "HELP!":LET X=0:
    GOTO 101
151 REM X>=0 HERE
```

The use of multiple commands per line also allows remarks to be placed on the line to which they apply. For example,

```
200 LET X=0:LET Y=0:LET Z=-1:
    REM INITIALIZE
```

Disk Operations

The micro disk system can be used by Basic 1 to save and recall programs and data. To accomplish this disk input and output, various strings of statements must be implemented.

The interpreter is the primary tool for the development of a Basic 1 Program. At the end of a programming session, the programmer might want to save the work for future reference. This interpreter allows the following string of Basic 1 statements, executed in the immediate mode, to permit the saving of a program:

```
DOUT Filename:LIST 10,20:PR "END":CLOSE
```

does not put an END statement on the program. The user must explicitly insert an END statement in the program before invoking DOUT to put the program on disk.

It should be noted that if a Basic 1 program is modified by use of the MicroDOS Resident Editor or if the program is written out without the keyword END in the command line, when the program is read into the interpreter, the error message 4 (misspelled statement type keyword) will be received.

The DOUT statement sets the disk as the primary output device and initializes the appropriate disk pointers by use of the associated parameter (file name). The LIST statement lists the program between the line numbers specified (the range of line numbers must be included). The PR statement prints the key word END after the listing. This key word, END, will be used when the program is again read in. Finally, the CLOSE statement puts a DC3 on the file, fills out the last sector, outputs it to the disk, and sets the console printer as the primary output device. An example:

```
DOUT HELP.BSC:1:LIST 1,1000:PR "END": CLOSE
```

A program can be read back into the interpreter with the following statement executed immediately:

```
RFLN
DIN READ?
Filename
```

In the immediate execution mode this statement will (1) set the disk as the primary input device, (2) initialize the appropriate disk pointers using the associated parameter (file name), (3) read lines from the disk to the input buffer, and (4) transfer lines preceded by a line number to the user area, executing lines not preceded by a line number. Thus, when the keyword END placed at the end of the file is read in, implicit TOUT and TIN statements are executed, setting the terminal and the console printer as the primary input and output devices. Example:

```
DIN RFLN
READ?
HELP.BSC:1
```

The output of the data to the disk is a relatively simple matter. First, the user must initialize the output file by means of one of the following statements:

```
100 DOUT Filename
100 WFLN
```

If a DOUT statement is used, the disk already has been set up as the primary output device. If the WFLN statement has been used, a DOUT statement with no parameter has to be used to set the disk as the primary output device. Once a file has been initialized and the disk set as the primary output device, a PRINT, PR, or LIST statement will output to the disk. When all the data is output, the output file must be terminated with the CLOSE statement, as discussed earlier. Some examples follow:

```
100....              100....
110....              110....
120....              120....
130 DOUT FILE:1      130 WFLN
140 PR "8,9,10"      140 DOUT
150 CLOSE            150 PR "8,9,10"
160 END              160 CLOSE:END
```

```
100....
110 DOUT FILE9
120 PR 8;
130 WEOF:DOUT:PR 9;
140 WEOF:DOUT PR 10
150 CLOSE
160 END
```

Inputting data from the disk is similar to outputting data. The first step is to initialize the input file by use of one of the following statements:

```
100 DIN Filename
100 RFLN
```

If a DIN statement was used, the disk has been set as the primary input device. If the RFLN statement was used, a DIN statement with no parameter must be used to set the disk as the primary input device. Once the file has been initialized and the disk set as the primary input device, any INPUT or IN statement will input from the disk. Examples follow:

```
100....              100....
110 DIN FILE1:1      110 RFLN:DIN
120....              120....
130....              130....
140....              140....
150 END              150 END
```

There are some rules regarding the format of data strings placed on a disk. These rules are:

1. All expressions in the input file must be separated by commas and the last expression must be followed by a carriage return.
2. Multiple lines with a carriage return and a DC3 separating them are allowed. (See WEOF statement.)

It should be noted that both the IN and the INPUT statements prompt the user with a question mark. This prompting will continue with disk input. To avoid the appearance of the question mark prompt on the console printer when a disk input is attempted, a dummy output file can be set up and the question mark prompt could be output as if it were data.

Basic 1 uses MicroDOS generalized I/O. The user can direct output to any device such as the screen, line printer, or disk. An example is:

```
DOUT #LP
```

#LP represents the line printer, #SC represents a twenty-two line screen, #TY represents a continuous console display.

Special Keyboard Control Characters

The Basic 1 interpreter allows certain key characters for deleting a character, cancelling a line, interrupting execution, and terminating a line.

An incorrectly entered character may be erased (backspaced over) by use of the "erase previous character" key. The hex code for this key is stored in location 0013 and is a RUBOUT. CONTROL H can also be used to delete a character from the screen and the input line buffer. Each occurrence of ← (*striking RUBOUT*) erases the last stored input character. In the event that a line has been butchered so badly that it is beyond repair, the entire line can be erased by use of the "cancel line" character. The hex code for this character is stored in location 0014 and is an ASCII cancel (*CONTROL C; hex03*). The user may change either of these edit control characters by changing its stored code to any value except DC3, LF, NULL, or DELETE (hex code 13, 0A, 00, and FF, respectively). These special characters are assigned by Basic 1 before its line code is entered.

The break key has a dual purpose in the interpreter. It can be used to interrupt a long listing or to interrupt the execution of a program (for example, one caught in an endless loop). While executing the list command, Basic 1 checks break at the beginning of every line. While executing a stored program, Basic 1 checks break between statements.

Each input line from the keyboard is terminated with a carriage return (CR).

A summary of the keyboard control characters is given in Fig. 8.

How the Compiler Uses the ! Statement

The Basic 1 compiler takes a source program written in Basic 1 and translates it into CDP1802 assembly language. Each line number is assigned a label, the label being the line number preceded by the letter L. When the compiler sees the ! statement in a program, it passes whatever follows the exclamation point to the output file as assembly language. No label is assigned to the line number associated with a ! statement.

A common use of the ! statement is to pass assembly language subroutines to the output file. A REM statement is commonly placed in front of the ! statement to provide a label for the user to call. Thus, if the user executes a GOSUB X, where X is the line number of the REM statement, the assembly language subroutine is executed, because REM statements are ignored during execution.

The ! statement is very important in that it allows the user to insert an assembly language subroutine anywhere in a program without having to be concerned with the absolute address of the subroutine as in the USR function and the GOSUB@ statement.

CR	Terminates every entry line
_ or ←	Backspace over or erase previous character
CAN	Cancel line
BREAK	Interrupt long listing or program execution

Registers R0, R1, R8, RA, RD, and RF are available to any assembly language subroutine. All other registers, excluding R2 (stack pointer), R3 (program counter), R4 (subroutine call), R5 (subroutine return), and R6 (return address storage), can be used if the previous values of these registers are saved at the beginning and restored at the end of the assembly language subroutine. The register values can be saved on the stack (R2 is the stack pointer). Assembly language subroutines receive control with P=3 and X=2 and must return with a SEP 5 instruction.

Loops and Subroutines

A loop is placed in a program to execute a number of statements over and over again until some condition is satisfied. An example follows:

```
100 REM COUNT DOWN
110 A=5
120 IF A<1 GOTO 160
130 PR "TIME MINUS",A
140 A=A-1
150 GOTO 120
160 PR "BLAST OFF"
170 END
```

Statements 120, 130, 140, and 150 are contained within the loop. Here the count down continues until the value of the variable A reaches zero.

A subroutine is a subprogram that is normally evoked two or more places within a main program. Rather than having the statements of the subprogram duplicated in several places, it appears only once. It is written so that it exists with a return statement. It is evoked at any point in a program by a GOSUB statement which transfers control to it.

Whenever one subroutine calls another subroutine (termed subroutine nesting), an additional "RETURN statement number" is recorded. These numbers are stored in order, so that every RETURN jumps back to the statement following the GOSUB which called it. Subroutines may be nested to any depth, limited only by the amount of user program memory remaining. It should be noted that GOSUB statements must be the last statement appearing on a line.

Programming Examples

The following programs are designed to give examples of Basic 1 in action. Remarks are omitted from the listings to keep them short. Instead, each program is accompanied by a detailed explanation of its functioning. It should be emphasized, however, that the omission of comments is generally poor documentation practice, but it suits the objectives of these examples. Each program can be entered in a few minutes. It is recommended that the user run them to gain experience with the system.

Arithmetic Drill Program

The arithmetic drill program generates a random sequence of arithmetic problems. After the program prints the problem, the user responds with a solution. The program tells the user whether the answer is correct or not (providing the correct answer in the latter case) and then proceeds to generate a new problem, and so on.

A listing of the program is given in Fig. 9. First, three random numbers are generated. The value of F (1 to 4) will be used to decide whether this problem will be an add, subtract, multiply, or divide problem. The range of possible values for the arguments A and B was chosen to prevent the possibility of overflow under two conditions. First, 181*181 is still less than 32767. Second, division by zero is prevented. Because Basic 1 discards division remainders, the fourth statement is included to keep the division problems interesting. It says: if this problem is a division problem in which the quotient would ordinarily come out as zero (true for many of the A, B, combinations that might be generated), arbitrarily increase the size of the dividend (to a maximum of 18100 in this case) to make the problem non-trivial. Statement 50 begins the presentation of the problem to the user by printing an encouraging message followed by the value of the first variable A. Notice that the final semicolon keeps the printer on the same line without advancing the carriage further.

Statement 60 does a four-way branch based on the value of F, the arithmetic function selected. Thus, control passes next to one of the following statement numbers: 70, 100, 130, or 160. Each of these statements begins a short sequence that prints the sign for the arithmetic operation and then computes the proper function, placing the result in C. Notice the final semicolons again in the print statements. No matter which path is taken, control passes next to statement 180, which prints the value of the second variable followed by an = sign. The presentation of the problem to the user is now complete, and the input statement at 190 delivers a ? prompt on the same print line and reads the user's answer into D. Statement 200 congratulates the user on a correct answer; statement 210 points out that his answer was incorrect and provides the correct answer. The commas at the end of both PRINT statements here again inhibit a new line from starting, but they space over to the next tab setting where a new problem is posed as a result of the loop at (220) back to the top.

It should be noticed in the program listing that an END statement is not present, contrary to earlier advice. The nature of the program is such that Basic 1

```
10      A = RND(1,181)
20      B = RND(1,181)
30      F = RND(1,4)
40      IF F = 4 IF A/B< 1 A = A*100
50      PRINT "TRY THIS ONE: ";A;
60      GOTO 40 + F*30
70      PRINT " + ";
80      C = A + B
90      GOTO 180
100     PRINT " - ";
110     C = A - B
120     GOTO 180
130     PRINT "*";
140     C = A*B
150     GOTO 180
160     PRINT "/";
170     C = A/B
180     PRINT B;" = ";
190     INPUT D
200     IF D = C PRINT "RIGHT!",
210     IF D< >C PRINT "WRONG. CORRECT ANSWER IS ";C,
220     GOTO 10
```

Fig. 9—Listing of arithmetic drill program.

will never go past the last statement. The program as written loops endlessly, and only under these conditions is the omission of an END permissible. However, if the program is to be compiled, an END statement must be placed at the end of the file to terminate compilation.

The running of this program should give the user some practice in learning how Basic 1 divides.

Geometric Print Pattern Program

This geometric print pattern program is designed to print three identical, trapezoidal patterns across the page, each filled with repeated imprints of the same number digit. The user can specify which digit is to fill each trapezoid and, for all three, the number of characters across its top, the slope of its sides (positive or negative), and its height. The user can also specify the spacing between the patterns on the page.

Because the printer prints line by line, the program prints the pattern in a scanning mode. Every line consists of a sequence of the three identical segments, and each segment contains D spaces followed by E identical digits followed by D spaces again. The values of D and E vary from line to line. For each new line, D is decremented by a value I (positive or negative) and E is incremented by 2*I (to keep the pattern symmetrical).

To analyze the program listed in Fig. 10, it is helpful to begin by identifying its subroutines. Reading from the bottom up, the subroutine from 250 to 280 prints the digit N across M times (notice the semicolon). Similarly, the subroutine from 210 to 240 prints a sequence of M spaces. Finally, the

```
10      J = 0
20      INPUT A,B,C,D,E,I,L
30      N = A
40      GOSUB 140
50      N = B
60      GOSUB 140
70      N = C
80      GOSUB 140
85      PRINT
90      D = D - I
100     E = E + 2*I
110     J = J + 1
120     IF J< >L GOTO 30
130     END
140     M = D
150     GOSUB 210
160     M = E
170     GOSUB 250
180     M = D
190     GOSUB 210
200     RETURN
210     PRINT" ";
220     M = M - 1
230     If M>0 GOTO 210
240     RETURN
250     PRINT N;
260     M = M - 1
270     IF M>0 GOTO 250
280     RETURN
```

Fig. 10—Listing of geometric print pattern program.

subroutine from 140 to 200 prints D spaces followed by E digits, all N, followed again by D spaces. Notice that this subroutine calls the other two.

The main part of the program runs from 10 to 130. First, the program initializes a counter J for the number of lines which have been printed. Then, it reads (from the user) initial values for A to E, I and L (the total number of lines to be printed). A, B, and C should be single digits, D, E, and L must be greater than 0. Each of these three sequence, 30-40, 50-60, and 70-80, prints one segment of a line using the digit specified by the user. A new line is started at 85. Statements 90 and 100 advance D and E as explained earlier, and 110-120 decide whether or not a sufficient number of lines has yet been printed. If not, a new line is started.

For this program to run properly, the values of D and E should not become to small. Nor should they be so large as to require excessive line length. The initial values should conform to the following relations:

$3(E + 2D) <$ maximum line width;
if $I < 0$, $E > 2|I|(L-1)$;
if $I > 0$, $D > I(L-1)$.

Built-in Subroutines for the USR Function

For convenience, the Basic 1 interpreter includes ~~four~~ built-in subroutines that can be used by means of the USR function. They are as follows:

```
(1) USR (#0114,N)        48;     LDA R8
    is a one-byte PEEK   D5;     RET
```

returns the decimal value of the byte at memory location N, where N is the value of the second expression.

```
(2) USR (#0118,N,M)      58;     STR R8
    is a one-byte POKE   D5;     RET
```

stores the value of the third expression, M(MOD 256), into the byte at location N, the value of the second expression. It also returns the value M as the function's value. Two examples follow:

PR USR(#0114,3072)

prints the decimal contents of memory location #0C00 (3072 = #0C00)

A = USR(#0118,#0C00,254)

loads memory location #0C00 with FE and also loads the returned value, 254, into A. (FE = 254)

(3) USR (#0106)

reads one ASCII character from the keyboard and returns its decimal equivalent (including parity bit, if any) + 256.

(4) USR (#0109, 0, C)

prints the ASCII character whose code is the right half of the hexadecimal value of expression C. Note that the second expression, in this case 0, is ignored. The character to be typed must start out in the D register; hence, the above format. The third expression is passed in RA with its lower half in the register D. This routine happens to return a value 251 in all cases, which would normally be ignored. Two examples follow:

PR USR(#0106)

will read a character and print its decimal equivalent. On the printer there would be, for example, A321 for a zero parity bit (where A was typed by the user).

A = A + 0*USR(#0109,0,66)

will print the character B and ignore the returned result (251).

```
(5) USR (#0144,N,M) is a two-byte POKE.  An example of
    its use is
        L=USR (#0144,#C000,P)

(6) USR (#0116,N) is a two-byte PEEK.  An example of
    its use is
        L=USR (#0116,#C002)
```

Error Messages and Program Debugging

Whenever the Basic 1 interpreter detects an error in a statement, it generates an error message consisting of an exclamation point followed by a decimal error number. A listing of the error numbers and their corresponding meanings is given in Appendix C. If the error is detected during program execution, the error code is followed by the word 'AT" followed by the number of the offending statement.

Almost all of the errors detected by Basic 1 are syntax errors. Basic 1 was in the process of interpreting a statement and found it unacceptable for some reason. Only two of the errors in the error list are detected during execution of a statement, i.e., after its syntax has been accepted, errors 141 and 243. Any other error number not listed in the table signifies a memory "full" condition, probably due to too many nested GOSUBS or an excessively complex expression.

Most program execution errors are due to either incorrect flow or improper modification of variable values. To find an error of the first kind, the user must determine whether the program is sequencing properly, i.e., whether certain sections of code are indeed executed when expected. Often, the insertion of dummy PRINT statements within suspected code

sections will reveal whether the flow within the program is proper.

The second type of error is most easily detected by the insertion of dummy program stops at key points. This procedure is also useful for diagnosing incorrect flow. A dummy stop is an inserted END, or some other inserted statement which is intentionally erroneous, to cause an error stop. Once the stop occurs, the values of key variables may be examined (using the immediate execution mode—e.g., PRINT A,B,C) to see if they indeed have the expected behavior. In some cases, variable values may be corrected, in the immediate mode, while the program is still stopped. In this case and in the case where the program behavior is correct so far, the user will want to resume the program at the point where it last stopped. An immediate or direct GOTO, using the statement number after the stop, will permit the program to proceed as if it had not been interrupted.

5. Running Basic 1 Compiler/Interpreter

The first step to be taken before anything can be done with Basic 1 interpreter, compiler, or any other program for that matter, is to power up the development system and run the utility program supplied with it. The Basic 1 interpreter and compiler also require the running of MicroDOS. It is assumed in this chapter that the user is already familiar with these procedures.

Loading and Starting the Interpreter

The Basic 1 interpreter is supplied on a floppy disk. To load the interpreter, the user should place the micro disk in one of the disk drives and type

BASIC1.INT:X

where X is the drive the floppy disk has been placed in. This command will load and start the interpreter. The interpreter will respond with a colon ":" prompt when it is ready to accept commands.

The interpreter can be restarted at any time by entering the MS2000 utility and typing in one of the following commands:

"P100" This command is for the normal "cold start". Basic 1 initializes itself and then delivers the colon prompt.

"P103" This command is for the "warm start", which skips the initialization procedure and preserves the state of memory. It also returns the colon prompt. The warm start is used when there is already a program in memory or when certain control parameters have been modified so that they are different from those which were first initialized. If, after a warm start, the user wishes to enter a new program he should type the command NEW.

Loading and Running the Compiler

The Basic 1 compiler is also supplied on a floppy disk. To load the Basic 1 compiler, the user should place the floppy disk in one of the drives and type

BASIC1.CMP:X

where X is the drive the disk has been placed in. This command will load and start the compiler. Execution will begin with the compiler prompting the user for the following:

LIBRARY?
READ?
WRITE?
ORIGIN?
VARS?
STACK?

The user responds to these prompts by typing a file name in the same form as discussed earlier for the library, read, and write files. The ORIGIN, VARS, and STACK prompts require the user to type the location in memory the program is to start, the location of the variables, and the location of the top of the stack. An example is given in Fig. 11 of the compiler prompts and a sample response.

If a carriage return is hit when the library file name is prompted, a default file name of BASIC1.LIB:0 is assumed. Any invalid file names will cause the printing of an error message followed by another prompt. The library file contains subroutines used by the compiler. After compiling the user program, the compiler calls in the desired subroutines from the library and places them in the write file. If the user

types a 46 (a key number) after the VARS prompt, the variables will be located in the same place as they were in the interpreter, as will TTYCC and the input line buffer (see Appendix D). The stack starts at high memory and works down. A minimum of 50 bytes should be saved for the stack; more should be saved for large programs. To correct errors, the user should type CONTROL-C for a new prompt.

Following is a further explanation with regard to the value given for a VARS? prompt issued by the compiler. When the user responds to the VARS? prompt with the value 46, storage is allocated for all variables that Basic 1 allows. The advantage of this scheme is that the user can return to the interpreter for debugging purposes and variables will be located where both interpreter and compiler have commonly allocated them. If a value other than 46 is given in response to VARS?, data is allocated for variables four bytes at a time for pairs of variables. For example, if only variable Y is used, the user will see in the output from the compiler:

```
        VY DC 0000H
        VZ DC 0000H
```

```
LIBRARY? BASIC1.LIB:1
READ? TEST1.SRC:1
WRITE? TEST1.CMP:1
ORIGIN? 256 or #100
VARS? 46 or #2E
STACK? #2FFF
```

Fig. 11—Compiler prompts and sample response.

Appendix A—
Formal Definition of Basic 1 Compiler/Interpreter

```
<PROGRAM>              ::= <STMTS> <CR> / <STORED PROGRAM>
<STORED PROGRAM>       ::= <LINE> / <LINE> <STORED PROGRAM>
<LINE>                 ::= <NSTRNG> <STMTS> <CR>
<STMTS>                ::= <LAST STMT> / <MORE STMTS> <COLON> <LAST STMT> /
                           <STATEMENT>
<MORE STMTS>           ::= <STATEMENT> / <STATEMENT> <COLON> <MORE STMTS>
<STATEMENT>            ::= LET@<EXPR> = <EXPR> /LET<VAR> = <EXPR> /<VAR> = <EXPR>/
                           PRINT<PRINTLIST> / PR<PRINTLIST> / *** "<STRING>" /
                           ***"<STRING>"<MOREPRINT>/
                           INPUT<INPUTLIST> / INPUT<INPUTITEM> / IN<INPUTLIST> /
                           IN<INPUT ITEM>
                           OUTPUT<EXPR> <COMMA> <EXPR> /
                           OUT<EXPR> <COMMA> <EXPR>/
                           ***LIST<EXPR> / ***LIST<EXPR> <COMMA> <EXPR>/
                           WEOF / TIN / TOUT / NOUT / DIN / DOUT
                           CLOSE / WFLN / RFLN / DOUT<FILENAME> / DIN<FILENAME>/< >
<LAST STMT>            ::= GOTO@<GOLIST> / GOTO<EXPR>/
                           GOSUB@<GOLIST> / GOSUB<EXPR>/
                           IF<CONDITIONAL>THEN <NUMBER> / IF<CONDITIONAL> <STMTS>/
                           IF <CONDITIONAL> THEN <STMTS>
                           *** LIST / RETURN / RET / END / *** STOP/ ***<RDOS> /
                           *** NEW / *** CLEAR / *** SCR / *** RUN<EXPRLIST>/ ***<RUN> /
                           REM<COMMENTS> / **** !<ASSEMBLY LANGUAGE> /
                           *** !<COMMENTS>/< >
<GOLIST>               ::= <EXPR> /<EXPR> <COMMA> <EXPR> /
                           <EXPR> <COMMA> <EXPR> <COMMA> <EXPR>
<PRINTLIST>            ::= <PRINTITEM> <MOREPRINT> / < >
<MOREPRINT>            ::= <SEPARATOR> <PRINTLIST>
<PRINTITEM>            ::= "<STRING>" / <EXPR> / HEX(<ARG12>)
<SEPARATOR>            ::= , / ;
<INPUTLIST>            ::= <VAR> / <VAR> <COMMA> <INPUTLIST>
<INPUTITEM>            ::= "<STRING>"<INPUTLIST> / "<STRING>"<COMMA> <INPUTLIST>
<CONDITIONAL>          ::= <EXPR> <RELOP> <EXPR>
<EXPRLIST>             ::= <COMMA> <EXPR> / <COMMA> <EXPR> <EXPRLIST>
<COMMA>                ::= ,
<ARG12>                ::= <EXPR> <COMMA> <EXPR> / <EXPR>
<EXPR>                 ::= <SIGN> <TERM> / <SIGN> <TERM> + <EXPR> /
                           <SIGN> <TERM> - <EXPR>
<TERM>                 ::= <FACTOR> / <FACTOR> *<TERM> / <FACTOR> ÷ <TERM>
<SIGN>                 ::= - / + /< >
<FACTOR>               ::= RND(<ARG12>) / AND(<ARG12>) / OR(<ARG12>) / XOR(<ARG12>)/
                           MAX(<ARGS>) / MIN(<ARGS>) / SGN(<EXPR>) / ABS(<EXPR>)/
                           MOD(<EXPR> <COMMA> <EXPR>) / USR(<GOLIST>) / INP (<EXPR>)/
                           <VAR> / <NSTRNG> / (<EXPR>) / @<EXPR> / #<HEXSTRNG>
<HEXSTRNG>             ::= <HEXNUMBER> / <HEXNUMBER> <HEXSTRNG>
<ARGS>                 ::= <EXPR> / <EXPR> <COMMA> <ARGS>
<RELOP                 ::= < / > / => / =< / = / <> / <= / >= / ><
<ASSEMBLY LANGUAGE>    ::= 1802 ASSEMBLY LANGUAGE
<FILENAME>             ::= <NAME> / <NAME> <FSUFFIX>
<NAME>                 ::= <LETTER> / <LETTER> <STRING>
<LETTER>               ::= A / B / C / ... / Z
<STRING>               ::= <CHAR> / <CHAR> <STRING>
<CHAR>                 ::= <LETTER> / <NUMBER>
<NUMBER>               ::= 0 / 1 / 2 / ... / 9
<FSUFFIX>              ::= <SUFFIX> / <UNIT> / <SUFFIX> <UNIT>
<SUFFIX>               ::= <SUFDEL> <SNAME>
<SUFDEL>               ::= .
<SNAME>                ::= <LETTER> / <LETTER> <CHAR> / <LETTER> <CHAR> <CHAR>
<UNIT>                 ::= <COLON> <NUMBER>
<NSTRNG>               ::= <NUMBER> / <NUMBER> <NSTRNG>
<COLON>                ::= :
<HEXNUMBER>            ::= 0 / 1 / 2 / ... / 9 / A / B / ... / F
<VAR>                  ::= <LETTER>
<COMMENTS>             ::= DOCUMENTATION COMMENTS
```

```
***   —INTERPRETER ONLY
****  —COMPILER ONLY
```

Appendix B— Summary of Basic 1 Repertoire

The following is a short-form guide to the facilities offered by Basic 1. Characters enclosed in brackets "[ ]" are optional and may be omitted. Commands marked with a single asterisk "*" are not accepted by the compiler. Commands marked with a double asterisk "**" have meanings that are slightly different to the interpreter and to the compiler.

Statements

FORM OF STATEMENT	BRIEF EXPLANATION OF EXECUTION
REM Any Comment	Ignored.
** ! Assembly Language	Interpreter—Ignored. Compiler—Pass assembly language to output file.
[LET] Variable = Expression	Assign the value of the Expression to the variable
LET@ Address = Datum	Stores a byte anywhere in memory.
GOTO Expression	Jump to the statement whose number is the expressions value.
GOSUB Expression	Save the statement number of the next statement. Then execute a GOTO.
GOSUB@ Exp1[,Exp2][,Exp3]	Jump to the machine language subroutine at address specified by Expression1. (Passes parameters specified by Expressions 2 and 3 if present.)
RET[URN]	Jump to the last saved statement number (see GOSUB) and "unsave" this number.
** END	Interpreter—halt execution and return to enter mode. Forces a TIN and a TOUT to be executed. Compiler—halt compilation.
IF Expr Rel Expr [THEN] Stmt	If the relation between the values of the expressions is true, execute the statement. Otherwise, skip it.
IN[PUT] Inputlist	Read and evaluate expressions from the keyboard and assign them in order to the variables specified in the input list.
PR[INT] Printlist	Type the items in the printlist. Type values of expressions. Type quoted strings verbatim. Horizontal tab on comma.
OUT[PUT] Port,Datum	Output the 8-bit data value to the specified port.
TIN	Restores terminal as the primary input device.
TOUT	Restores console printer as the primary output device.

NOUT	Disables output.
DIN [Filename]	Sets internal flags for disk input. Closes previous input file and opens new file number specified filename (if present).
DOUT [Filename]	Sets internal flags for disk outout. Closes previous output file and opens new file under specified filename (if present).
WEOF	Writes an end-of-file character (DC3) on the output disk file last referenced. Then executes a TOUT.
WFLN	Reads filename from terminal, closes previous output file, and opens new file under specified filename. Then executes a TOUT and a TIN.
RFLN	Reads filename from the terminal, closes previous input file, and opens new file under specified filename. Then executes a TOUT and a TIN.
CLOSE	Closes previous output file. Executes a TOUT.
* NEW or * CLEAR or * SCR	Clears the program area.
* RUN [,Expression Sequence]	Start execution at first statement. (Save the expression sequence to satisfy subsequent IN commands.)
* LIST [Expr][,Expr]	Prints entire program or one selected line or a range of lines.
* RDOS	Returns control to MicroDOS.

FUNCTION	PERFORMANCE
MOD (Exp1,Exp2)	Divides Exp1 by Exp2 and returns remainder.
AND (Exp1,Exp2)	Returns the bit-for-bit logical AND of the two expressions.
OR (Exp1,Exp2)	Returns the bit-for-bit logical OR of the two expressions.
XOR (Exp1,Exp2)	Returns the bit-for-bit logical XOR of the two expressions
MAX (Expression Sequence)	Returns the greatest value in the expression list.
MIN (Expression Sequence)	Returns the smallest value in the expression list.
SGN (Expression)	Returns A −1, 0, or +1 depending on whether the expression is negative, zero or positive, respectively.
ABS (Expression)	Returns the absolute value of the expression.

HEX (Number, Width)	Used only in PR[INT] statements. Prints value of the number in hexadecimal. Both the number and the width can be any valid expression.
RND (Exp1,ExP2)	Returns a positive random number in the range between the values of the expressions.
INP(Port)	Inputs an 8-bit value from the specified port.
USR (Exp1[,Exp2][,Exp3])	Used for machine language subroutine call, passing parameters Exp2, Exp3. Returns value from certain registers.

WHERE:

Number = −32768 TO 32767

Variable = Single Capital Letter

Expression = One or more numbers, variables or functions (possibly grouped by parentheses) joined by arithmetic operators +, −, *, / or the indirection operator @.

Relations are =, >, <, < =, > =, < >, > <, = >, or = <

Printlist = One or more expressions or quoted strings separated by commas.

Inputlist = One or more variables separated by commas.

Expression Sequence = One or more expressions separated by commas.

Filename = Character string in the form: (NAME)[.EXTENSION][:DRIVE]
The first character of the name and extension must be an alphabetic character.

Appendix C—
Summary of Error Messages

0	Break during execution
2	RETURN has no matching GOSUB
3	No END statement
4	Misspelled statement type keyword
5	Syntax error
6	Syntax error
7	GOTO or GOSUB error—no line to go to
9	Missing right parentheses
10	Syntax error
11	Missing statement type keyword or number outside range −32768 to 32767
13	Divide by zero
14	Syntax error—missing variable name
15	Syntax error—missing =
16	IF expects relational operator
22	Line number too large (greater than 32767)
23	Missing close quote in PRINT string
26	Line number 0 not allowed
29	Run with no program in memory
318	Can't LIST line number 0
1001	Invalid arguments in RND

Note: In some cases multiple errors may have a masking effect resulting in either no error being flagged or an erroneous error number being printed.

Appendix D— Scratchpad Memory Locations in the Interpreter

PAGE	= #0000	..Beginning of work space
TIO	= #0012	..TTY timing flag
BS	= #0013	..Location of rubout code
CAN	= #0014	..Location of cancel code
TAPE	= #0016	..Location of tape mode enable
SPARE	= #0017	..Spare stack space
XEQ	= #0018	..Execution mode flag
LEND	= #0019	..Input line end
AEPTR	= #001A	..Expression stack pointer
TTYCC	= #001B	..Print column counter and flag
NXA	= #001C	..Saved PC for (subroutine NXT)
AIL	= #001E	..Address of intermediate level language
BASIC	= #0020	..Address of user code
STACK	= #0022	..Address of memory top
MEND	= #0024	..Address of program end
TOPS	= #0026	..GOSUB stack top
LINO	= #0028	..Basic program current line #
WORK	= #002A	..4 bytes of scratch
SP	= #002E	..Saved pointer
LINE	= #0030	..Input line buffer
AESTK	= #0080	..End of alternate expression stack and line buffer
VARS	= #0082	..Beginning of variables

Low Memory Map in the Interpreter

0100	..Cold-start entry point
0103	..Warm-start entry point
0106	..Long branch to character input
0109	..Long branch to character output
010C	..Long branch to break test
010F	..Rubout code
0110	..Line cancel code
0111	..Pad character
0112	..Tape mode enable flag
0113	..Spare stack size
0114	..Subroutine to read one byte from RAM to RA
0118	..Subroutine to store RA into RAM at address in R8
011C	..User address constant
1600	..Beginning of user program

Appendix E— ASCII—Hex Table

MOST SIGNIFICANT HEX DIGIT

LEAST SIGNIFICANT HEX DIGIT	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	¦
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	↑	n	~
F	SI	US	/	?	O	←	o	DEL

NOTES:

(1) Parity bit in most significant hex digit not included.

(2) Characters in columns 0 and 1 (as well as SP and DEL) are non-printing.

(3) Model 33 Teletypewriter prints codes in columns 6 and 7 as if they were column 4 and 5 codes.