# MONROE BASIC
# PROGRAMMER'S REFERENCE MANUAL

February 1982

MONROE SYSTEMS FOR BUSINESS
The American Road
Morris Plains, N.J. 07950

Except as stated in the license agreement for this software, Monroe does not warrant this software or its documentation, either expressly, by implication or in relation to merchantability or fitness for a particular purpose.

Monroe shall not be liable for any incidental, indirect, special, consequential, or punitive damages arising out of or in any way connected with the use, furnishing of or any failure to furnish software or any related materials, including, but not limited to, claims for lost profits, increased expenses or costs, loss of good will, or damage to property. This exclusion of liability shall apply without regard to whether such damages were foreseeable or foreseen or are claimed to arise by reason of breach of contract, breach of warranty, misrepresentation, negligence, strict liability, or other legal theory.

Monroe reserves the right to make changes in the content of this software or its documentation without obligation to notify customer of such changes.

## PURPOSE OF THIS DOCUMENT

This document is a Programmer's Reference Manual. It is to be used by experienced programmers as a reference tool. It is not intended for use as a learning aid by non-programmers.

# RECORD OF CHANGES

| Change No. | Date | Pages Affected | Description of Change |
|---|---|---|---|
| -4 | 7/81 | | Preliminary Edition |
| -5 | 2/82 | | First Edition |

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

LIST OF TABLES

LIST OF TABLES (Cont.)



LIST OF ILLUSTRATIONS

# SECTION 1
# INTRODUCTION

# SECTION 1
# INTRODUCTION

## 1.1 OVERVIEW

This manual describes Monroe's Extended BASIC hereafter referred to as Monroe BASIC. Monroe BASIC is a comprehensive, semi-compiled language which is available on the Monroe 8800 Computer Series. It is an implementation of the BASIC language initially developed by Dartmouth College and standardized by the American Standards Institute (American National Standard for Minimal BASIC, ANSI X.360-1978).

A newcomer to programming and computers should read Monroe's introductory texts on the Monroe BASIC language and system and peripheral operation. It must be emphasized that this is a Programmer's Reference Manual and not a tutorial. Hence, it is designed to be used primarily as a reference device by experienced programmers.

BASIC is one of the simplest of all programming languages because of its small number of powerful but easily understood statements, functions and commands and its easy application to problem solving. Nevertheless, the language is comprehensive enough to allow versatile and efficient solutions to most problems. The wide use of BASIC in scientific, business, and education installations attests to its value and straightforward application.

Monroe BASIC is an interpreter program stored on disk and called into memory by the user when required. An interpreter is a type of compiler which checks or interprets your source program as you enter it line by line. The source program resides in memory along with the interpreter for as long as the user requires and can be saved and run whenever needed. This is in contrast to other compilers which save the computer readable form (i.e. the object program) and then execute the object program whenever needed. Because Monroe BASIC is an interpretive language, a syntax error will result in an immediate error message on the screen. You can also run the program at any time to test portions that have been entered. This is called interactive programming and is in many cases the most efficient way of programming. However, interactive programming does not solve all

the problems. When formal errors have been eliminated from the program, logical errors may still remain. These can only be detected when the program is executed with the proper data.

Monroe BASIC contains elementary statements to write simple programs. Advanced programming features and statements are also provided to produce more complex and efficient programs. The keyword here is efficient. Almost any problem can be solved with the simple Monroe BASIC statements. Later in the user's programming experience, the advanced techniques can be added. Monroe BASIC also allows the use of multi-character variable names and free use of comments and spaces, which aid in creating programs that are self-documenting and maintainable.

## 1.2 TEXT SYMBOLS AND CONVENTIONS

Throughout this manual specific documentation conventions are used to describe formats for writing Monroe BASIC commands, statements, and functions. The following conventions are in effect:

| Symbol | Description and Use |
|---|---|
| 1. CAPITAL LETTERS | Capital letters are used for all keywords, commands, functions, and statements that are to be explicitly typed.<br>Example: LIST |
| 2. Lower case | Lower case letters specify variables which are to be supplied by the user according to the rules explained below and in this text.<br>Example: DATA <list> |
| 3. < > | Angle brackets enclose fields that are required for valid Monroe BASIC syntax. They are never to be typed unless "not equal to" is to be specified by "<>".<br>Example: LET <variable> = <express><br>LET A = 4 |

| Symbol | Description and Use |
|---|---|
| 4. ( ) , - : | Parentheses enclose required elements or keywords of a statement. Commas, dashes and colons are separators. All must be typed as shown.<br>Examples: COMP%(A$,B$): DIM C$(-3:6) = 200<br>LIST 100-400 |
| 5. [ ] | Square brackets enclose optional elements of a statement or indicate an optional choice of one element among optional elements.<br>Example: GET <stringvar> [COUNT bytes] |
| 6. ... , ... | Ellipsis (three dots) indicate multiple arguments are allowed.<br>Example: POKE <address>,<data>[,data,...,...] |
| 7. ¶ | The symbol "¶" indicates the depression of the RETURN key.<br>Example: LIST¶ |
| 8. CTRL-H | Control character. Depress and hold CTRL key while striking another key (represented by H). |

## 1.3 FILE-VOLUME-DEVICE NAMING CONVENTIONS

The Monroe Operating System file, volume, and device naming conventions are defined as follows:

A) A file is a program or a collection of data stored on a disk-type storage medium. Once saved, files stay on the disk permanently unless they are explicitly removed.

B) A volume name is a name given by the user to a disk. Filenames must be preceded by their volume names unless they reside on the system volume. The system volume is the volume from which the operating system is booted. It can be changed by the user.

C) A device name is a name given to a physical device (e.g., CON: for the console, PR: for the printer, FPY0: for drive 0 (lower drive), FPY1: for drive 1 (upper drive)). These names cannot be changed by the user.

D) File descriptors, hereafter referred to as "fd" in this manual, can be composed of up to four fields: vol, filename, directory, and type, where vol can be either a volume or, when used alone, a device name. Device descriptors are composed of the device mnemonic only.

E) The format can be expressed in the following ways:

1. <device:>
2. [vol:][directory.]<filename>[/type]

where:

vol/device — Vol is the name of the disk on which the file resides if the file descriptor refers to a file, or the name of a device if the file descriptor refers to a device. It may be from one to four characters. The first character must be alphabetic and the remaining alphanumeric. If the volume is not specified, the default volume is the SYSTEM volume. Note that a minus sign (-:) can be specified in place of the actual volume name. Then, in searching for an existing file, the system will automatically search for all mounted volumes. In order to ensure that a new file will reside on the correct volume, a default or "-" volume name should not be used when creating files.

filename — Name of the file. It may be from one to twelve characters, the first alphabetic and the remaining alphanumeric.

directory — Name of the user's file directory. It may be from one to twelve alphanumeric characters. If not specified, the directory defaults to the master directory.

type — A single letter which describes the type of data within a file. For the Monroe BASIC commands SAVE, UNSAVE, LOAD, CHAIN and RUN, the system recognizes two different types:

B (default) BAC-BASIC compressed form
AB (ASCBAS) - BASIC uncompressed ASCII form

When the file type is omitted for the above Monroe BASIC commands the computer will look for type BAC (compressed form) first and then ASCBAS (uncompressed form). If type "AB" (ASCBAS) is specified only ASCBAS will be searched for in the directory.

For the Monroe BASIC statements OPEN, NAME, KILL, MERGE, LIST and PREPARE the default is "AB" if type is not specified.

Refer to the 8800 Series Monroe Operating System Programmer's Reference Manual for a detailed explanation of type.

Examples:

Examples of legal file descriptors and their usage are:

| | |
|---|---|
| LIST PR: | The current program is displayed on printer. |
| MERGE MAIN | Merges lines from ASCII file MAIN from the Master Directory into the current program. |
| LOAD PACK:MAIN | Loads program MAIN from disk "PACK" into working memory. |
| UNSAVE PACK:DIRECTA.MAIN | Deletes program MAIN from user directory "DIRECTA" on volume "PACK". |

## 1.4 CHARACTER SET

Monroe BASIC is designed to utilize the American Standard Code for Information Exchange (ASCII) for its character set. This set includes:

1. Printable Characters-
   a. Letters A through Z
   b. Lower case letters a through z
   c. Numbers 0 through 9
   d. Special punctuation characters and symbols
2. Control Characters

Lower case letters are treated equivalently to upper case letters by Monroe BASIC except in strings, DATA and REMARK statements.

Appendix A shows the complete set of Monroe BASIC ASCII characters and their respective decimal codes.

## 1.5 ORGANIZATION OF THIS MANUAL

This manual is organized into 15 sections and seven appendices.

Sections 2 through 5 contain information necessary to understand and work with Monroe BASIC. There are many types of BASIC in use today. Hence, these sections define the procedures and language elements within the context of Monroe BASIC language.

Sections 6 through 10 describe the individual control commands, statements, and functions available in Monroe BASIC. Statements are divided into three sections: Data, Input/Output and Program Control statements for ease of reference. For each command, statement, or function, the following information is included:

1. Function - Summarizes purpose of statement.
2. Mode - Specifies which mode applies - Direct, Program or both.
3. Format - Shows the command syntax.
4. Arguments - Defines the format variables.
5. Use - Describes in detail how the command is used including restrictions and exceptions.
6. Example - Lists program examples illustrating the various uses of the command.

Section 11 describes how the appearance of output data can be controlled using the PRINT USING statement.

Sections 12 and 13 deal with low and high resolution color graphics, respectively.

Section 14 contains advanced file information and Monroe BASIC statements and functions to be used for sophisticated programming. Detailed knowledge of Monroe BASIC and the operating system is required before this information can be applied.

Section 15 describes low resolution business graphics.

Appendix A shows the complete set of Monroe BASIC ASCII characters and their respective decimal codes.

Appendix B contains a list of error messages with comments.

Appendix C contains sample programs.

Appendix D lists the port numbers and associated devices.

Appendix E shows the low resolution character set.

Appendix F contains the high resolution color selection table.

Appendix G contains a quick reference summary of Monroe BASIC commands, statements, and functions.

## 1.6 ABBREVIATIONS

The following abbreviations are used in this manual:

| | | |
|---|---|---|
| channel no. | - | Channel Number |
| fd | - | File Descriptor |
| line no. | - | Line Number |
| record no. | - | Record Number |
| string var | - | String Variable |
| vol | - | Volume |

## 1.7 RELATED MANUALS

This document is to be used as a reference manual. If more instructional information is needed refer to the Monroe BASIC Primer.

If additional information on the operating system's software is required refer to the following 8800 Series Programmer's Reference Manuals:

- Monroe Utility Programs
- Monroe Text Editor
- Monroe Operating System

# SECTION 2
# WORKING WITH MONROE BASIC

SECTION 2

WORKING WITH MONROE BASIC

## 2.1 INITIATING AND TERMINATING MONROE BASIC

Monroe BASIC is delivered on a diskette. To load Monroe BASIC, enter the word "BASIC" in response to the operating system prompt "-". Optional parameters may be included after "BASIC" as shown below.

```
-BASIC[,[C][,xmemory]]¶   [Note: ¶ = Return key]
```

Where: xmemory is optional. It is the additional memory size required in bytes to load or to enter the desired program. Since memory is allocated in blocks of 8,192 bytes; xmemory is rounded up to the next multiple of this value. The minimum amount always provided is 6600 bytes.

C is an optional switch which when specified disables the STOP key (or the CTRL-C sequence). This prevents the user from stopping the execution of a program once it begins. (This switch is present only in Monroe BASIC versions R1-03 or later.)

Monroe BASIC begins loading and when ready displays the following:

```
BASIC8 Rn-xx yyyy-mm-dd   (where Rn-xx yyyy-mm-dd refers
BASIC                      to the Version Level and Rev.
                           Date)
```

It is also possible to execute a Monroe BASIC program directly from the operating system by entering:

```
-BASIC[,[C][,xmemory]] <fd>¶   (where fd refers to the file
                                descriptor as previously
                                defined in Section 1.3)
```

More than one Monroe BASIC program can be run simultaneously. The LOAD and START commands necessary to do this are shown in Appendix C and described in the 8800 Series Utility Programs Programmer's Reference Manual. The method of starting one Monroe BASIC program from another is also shown in Appendix C.

To remove Monroe BASIC from working memory and return to the operating system type:

```
BYE¶
-
```

The operating system prompts with a dash (-) for the next series of commands.

## 2.2 MODES OF OPERATION

Monroe BASIC allows for three modes of operation:

1. Program Mode
2. Direct Mode
3. Run Mode

### Program Mode

Monroe BASIC distinguishes between those statements intended for immediate execution and those for delayed execution. This difference is based solely on the absence or presence of a line number in front of a statement. A statement preceded by a number, for example:

```
180 INPUT X,Y
```

is noted as being intended for delayed execution. Statements discussed in this text to which this definition is applicable have "Program" specified after the Mode declaration.

### Direct Mode

Conversely, the absence of a line number in statements such as:

```
PRINT 1000/12¶
```

cause the interpreter program to execute it directly after depression of the RETURN key. This is called the direct mode and so specified for applicable statements in Sections 6 through 14 in this text.

The Direct mode also allows for immediate solution of problems, generally mathematical, which do not require interactive program procedures. For example:

```
A=1.5: B=3: PRINT A, B, "ANS-"; (A+B*A)
```

Another use of the direct mode is as an aid in program development and debugging. Through use of direct statements, program variables can be altered or read, and program flow may be directly controlled.

Direct statements in combination with program variables can be used whenever the BASIC edit mode prompt appears or

-after CTRL-C has been typed
-after an error message has been printed
-after a STOP statement has been executed
-after a program terminates normally.

Run Mode

A program consisting of a series of numbered statements can be executed only in the Run Mode. Execution (run mode) is begun when the RUN command is entered or the RUN key is typed: For example:

```
80 A = 5¶
100 PRINT A¶
RUN¶
5
BASIC
```

## 2.3 PROGRAM STRUCTURE

A user program is composed of one or more properly formed Monroe BASIC statements, constructed with the language elements and syntax described in the following sections. A statement contains instructions to Monroe BASIC. A program line begins with a line number followed by one or more Monroe BASIC statements, up to a maximum of 157 characters. Line numbers indicate the particular sequence of execution. Each statement begins with a keyword specifying the type of operation to be performed. A program line can also contain multiple statements.

Each statement gives an instruction to the computer (in this example PRINT):

```
30 PPINT S
```

The value currently assigned to the variable "S", above, is printed. If the instruction requires further details, operands (numeric details) are supplied. The operands specify what the instruction acts upon, (for example, GOTO):

```
40    GOTO 10
```

In the above example, the operand "10" is the line number to which program control will be transferred upon execution of the "GO TO" statement.

The last statement in a program, as shown here, is an END statement.

```
10    INPUT A,B,C,D,E
20    LET S = (A+B+C+D+E)/5
30    IF A=999 GOTO 60
40    PRINT S
50    GOTO 10
60    END
```

The END statement informs the computer that the program is finished, but its presence is not mandatory.

## 2.4 LINE NUMBERING

Each program line in the program mode is preceded by a line number. A line number has the following effects.

1. Indicates the order in which the statements are executed. (The statements may be written in any order.)

2. Enables the normal order of evaluation to be changed by GOTO, GOSUB statements, etc.

3. Permits program modification of any specified line without affecting any other portion of the program.

The line number is chosen by the programmer. It may be any integer from 1 to 65,535 inclusive. The system uses the line numbers to keep the program lines in order and for the execution required.

Program lines may be entered in any order; they are usually numbered by fives or tens so that additional statements can be easily inserted. The computer keeps them in numerical order no matter how they are entered. For example, if the program lines are input in the sequence 30, 10, 20, Monroe BASIC rearranges them in order: 10, 20, 30. These are commands for automatic line numbering (AUTO) and for renumbering (REN).

## 2.5 STATEMENTS

A program line begins with a line number followed by a Monroe BASIC statement. The keyword of a Monroe BASIC statement identifies the type of statement. Monroe BASIC is thereby informed what operation to perform and how to treat the data - if any - that follows the keyword.

### Multiple Statements on a Program Line

The user is allowed to write more than one statement on a single line. A maximum of 157 characters per line is permitted. Each multi-statement (except the last) is terminated with a colon. Only the first statement on the program line can have a line number preceding it.

Example:

```
100 PRINT A,B,C
```

is a single statement program line.

```
200 LET X=X+1 : PRINT X : IF Y = 1 GOTO 100
```

is a multiple statement program line containing three statements: LET, PRINT and IF-GOTO.

As a rule any statement can be used anywhere in a multiple statement line. The exceptions to the rule have been explicitly specified in individual statement descriptions.

## 2.6 LINE ENTRY

After calling Monroe BASIC, the following default modes are initially in effect:

| | |
|---|---|
| NO EXTEND | Variable names can only be composed of one letter and an optional digit. |
| FLOAT | Numbers and variables without the "%" suffix are interpreted as floating point. |
| SINGLE | Precision of variables or expressions is accurate up to six significant digits. |

In the default NO EXTEND mode, Monroe BASIC is a "free format" language - it ignores most blank spaces in a statement. For example, these four statements are equivalent:

```
30  PRINT S
30  PRINT   S
30PRINTS
30P RINT S
```

When a program is listed, Monroe BASIC adds spaces to make the listing more readable. It is important to note that entered spaces are significant in the following areas:

1. REM Statements and Comments
2. String Constants
3. Data Statements

### Procedure

Lines input to Monroe BASIC are either executed immediately (Direct Mode) or stored in the user program area for later execution (Program Mode). Program mode statements can also be saved on disk for future execution. Monroe BASIC accepts lines when it is not executing a program. The RETURN ¶ key must be pressed after each line.

Example:

```
10  INPUT A,B,C,D,E¶
20  LET S=(A+B+C+D+E)/5¶
30  PRINT S¶
40  IF A <= 999 GOTO 10¶
50  END¶
```

Pressing RETURN "¶" informs Monroe BASIC that the line is complete. Monroe BASIC then checks the line for mistakes in syntax. If such mistakes are found, an error message is displayed on the screen. (Refer to Section 2.7.)

Immediate Corrections

The row of terminal keys at the top right of the keyboard, the CTRL key and the cursor left and right keys can be used for immediate corrections. Immediate corrections can be made to a program line or to the data entered in response to the execution of an INPUT or INPUT LINE statement. Operation begins in the overstrike mode in which each character typed replaces the character over the cursor. The cursor is then moved to the right one character. The following correction keys are available:

| | |
|---|---|
| BACKSPACE | In overstrike mode replaces the character to the left of the cursor with a blank and moves the cursor one position to the left. In insert mode (see below), deletes the character to the left of the cursor and moves the cursor one position to the left. |
| DELETE LINE | Causes all the text on the line to be deleted. |
| CURSOR LEFT (←) | Moves the cursor one position to the left for each touch of the key. |
| CTRL-← | Moves the cursor to the beginning of the entered line. |
| CURSOR RIGHT (→) | Moves the cursor one position to the right for each touch of the key. |
| CTRL-→ | Moves the cursor to the end of the entered line. |
| DELETE CHAR | Deletes the character at the current cursor position and moves all the following characters one position to the left. |
| INSERT LOCK | Allows characters to be inserted into the text string to the left of the character where the cursor is positioned. Insert mode is cancelled by ¶ or any cursor key. |

RETURN — Terminates corrections. Cursor does not have to be positioned at the right margin when RETURN is entered.

## Deleting a Statement

To delete the statement being typed, depress CTRL X or the LINE DEL key. This deletes the entire line being typed. For example:

```
20 LET S = 12X CTRL X
```

To delete a previously typed statement, type the statement number followed by a RETURN "¶". For example:

```
5     LET S = 0
10    INPUT A,B,C,D,E
20    LET S = (A+B+C+D+E)/5
```

To delete Statement 5, above, type:

```
5¶
```

To delete blocks of statements, refer to ERASE, Section 7.

## Changing a Statement

To change a previously typed statement (in program mode), retype it with the desired changes. The new statement replaces the old one.

To change statement 5 in the above sequence, type:

```
5 LET S = 5¶
```

The old statement is replaced by the new one. Use the LIST command to check what is left of the program.

If a single or a few characters need to be corrected, use the ED (EDIT) command. (Refer to Section 6.5.)

Blocks of statements from another program can be inserted by the MERGE commands. The user thereby has the possibility to handle programming on a modular basis.

It is important to note that when a statement is changed, an implicit CLEAR command is performed.

## 2.7 EDITING A PROGRAM

Lines may be deleted, inserted or changed according to the procedures described previously in this section and the commands that are available in Monroe BASIC. The LOAD command places the desired program into working storage. The MERGE command allows you to combine or change your program with a set of statements loaded from a disk file. The ERASE command deletes blocks of statements. The ED command facilitates corrections of an existing line on a character basis.

When editing a program, you may want to increase or decrease increments between selected lines. This is done by the RENUMBER Command after additions or deletions have been done.

If there is a syntax error in the previously typed statement, an error message is printed. This line is not entered but is retained in a save area and displayed on the screen with the cursor positioned to the last character. The erroneous line may be immediately edited using the same set of special keys as specified for the EDIT command. (Refer to Section 6.5.)

Example:

```
10 PRONT CUR (2,30) "TROUBLE REPORT"¶
UNDECODEABLE STATEMENT
10 PRONT CUR (2,30) "TROUBLE REPORT"_
(Use cursor left (←) to position cursor over
the 0 in pront, enter an "I" and then depress
RETURN)
LIST 10¶
10 PRINT CUR(2,30) "TROUBLE REPORT"
BASIC
```

## 2.8 EXECUTING A PROGRAM

The RUN command is provided to start the execution of a program. When the command (RUN ¶) is entered, Monroe BASIC starts to execute the program in the user's program area at the lowest numbered line. Execution continues until either one of these conditions is encountered:

STOP
END
ERROR

When the program executes a STOP or END statement it halts and all the variables are still in existence. The user can examine the variables by simply addressing the respective ones by the variable name. For example, you want to know the values of the variables A, S, and K%. Enter the following command:

```
PRINT A,S,K%¶
```

The computer will then write the current values of the variables when program execution was stopped.

Errors cause an error message to be written on the screen. See Appendix B for the complete set of error messages.

A running program can be halted by typing CTRL C (both keys simultaneously) or by typing the STOP key. After halting, it is possible to single step the program by depressing and holding the CTRL key and then simultaneously striking the S key once for each step. To continue execution depress any key. The program can be stopped again as specified above.

## 2.9 DOCUMENTING A PROGRAM

Monroe BASIC permits the programmer to document a program with notes, comments and messages. There are two methods available: Standard REM statements and text preceded by an exclamation point. The latter type of comments are easier to use since they can occur without a colon.

Examples:

a. 10 A = 7: REM ASSIGN "7" TO THE VARIABLE "A"

b. 10 A = 7! ASSIGN "7" TO THE VARIABLE "A"

REM lines are part of a BASIC program and are printed when the program is listed; however, they are ignored when the program is executing. Any series of characters may be used in a comment line. The remarks are usually marked with some clearly visible character, making them easily noticed in a program. For example:

```
100 REM*** CAUTION ***
```

A comment cannot be terminated by a colon. The colon is treated as part of the remark. For example:

```
150   REM ***INITIALIZE R1***:LET R1=3.5E2
```

The assignment statement will not be executed. The entire line is considered to be a non-executable comment.

Indentation is another method of documentation. Any spaces entered between the line number and the first character of a line will be ignored by Monroe BASIC. Monroe BASIC automatically indents FOR/NEXT loops, WHILE loops and multiple line user defined functions.

## 2.10 FILE USAGE

Monroe BASIC provides facilities to define and manipulate input and output data on the disk drives, console, printers and other non file-structured devices. Three file access methods are supported: Sequential (one record after another from beginning of the file), Random (by relative record number), and Indexed Sequential (random by key). Random and Indexed Sequential access are discussed in Section 14, "Advanced Programming".

A data file consists of a sequence of data items transmitted between a BASIC program and an external input/output device. The external device can be the user's terminal, printer or disk.

Each data file is externally identified by a file name, (e.g. ADC123). Internally in the user's program, the file is accessed as a channel number. PREPARE, OPEN and CLOSE statements are used to establish and terminate a channel for the data transfer. All further references to the file in the program will be to channel number (e.g. #1) not to file name - ADC123.

Random I/O permits the user's program to have complete control of I/O operations. Properly used, Random I/O is the most flexible and efficient technique of data transfer available under Monroe BASIC. It is, however, not as simple as Sequential I/O. Less experienced users should first experiment with the Sequential I/O techniques before attempting Random I/O. Random I/O is explained in detail in Section 14, Advanced Programming.

The file number is given in the program by means of one of the instructions PREPARE or OPEN. These statements will open the file, i.e. set up a channel for the data transfer. To explicitly close such a data transfer channel the instruction CLOSE is used. Files are automatically closed when a BASIC program terminates normally. The instructions INPUT and PRINT or GET and PUT are used for the data transfer.

A buffer area is created by the system when a file is opened. All data transfer to and from a file is buffered. Channel number 0 always refers to the console; attempts to open or close channel 0 are ignored.

Opening a File

To open an existing file the OPEN statement is used. If the file is new, it should be opened with a PREPARE statement.

Example:

```
10 OPEN "MAST" AS FILE 1
```

opens existing file named MAST for input/output and assigns logical unit 1, for I/O, to that file.

Data Transfer To/From a File

The transfer of data takes place directly between the internal channel (the file number) and the string variable or the value of the expression in question.

The following instructions can be used:

| | |
|---|---|
| INPUT # | Reads a value to a variable or a string from the position of the file pointer to a carriage return. |
| INPUT LINE # | Reads a value to a string variable including the carriage (CR) return and line feed (LF). Also accepts leading, trailing, and embedded spaces and commas. |
| PRINT # | Writes the contents of a variable into the file. |
| GET # | Reads one byte from the position of the file pointer. |
| GET # COUNT nn | Reads the given number of bytes from the position of the file pointer. |
| PUT # | Writes one record into the file. |
| POSIT # | Moves the file pointer to the desired position. |

If no file number is given in the GET statement, it will attempt to read from the keyboard.

Example:

```
20 GET #1,D$ COUNT 6%
```

will read from file number 1 the first six characters starting from the position of the file pointer. . These characters are put in the string D$.

The instruction POSIT is used to position the file pointer at the given position in the file. The number of characters always refers to the beginning of the file (position 0). POSIT can be used together with any one of the other file handling instructions.

Example: LIST¶

```
40 OPEN "PEARL" AS FILE 1 ! PEARL contains ABCDEFGHIJK.
50 POSIT #1,5
60 GET #1,A$ COUNT 3
70 PRINT A$
80 ; POSIT (1)¶
90 END¶
RUN¶
FGH
8
BASIC
```

The function POSIT(<file number>) reads the position of the file pointer. In the example above, POSIT(1) returns the value 8, when the example has been executed. POSIT returns a floating point value, so that very long files can be handled.

## Closing a File

The data transfer to or from a file will not be correctly terminated until the file is closed. The contents of the buffer area are then transferred, and the file is given an end-of-file (EOF) mark.

There are two ways of explicitly closing a file:

1. CLOSE n — closes the file associated with file number n
2. CLOSE — closes all files

Also, all files are automatically closed when the END statement of a BASIC program is executed. Files are also closed whenever a program is modified.

## 2.11 LOGICAL UNITS

Monroe BASIC ensures independence from physical input/output devices through the use of file numbers. The file number can be treated as a logical unit and is handled with the instructions OPEN, PREPARE and CLOSE. The file number may, for instance, represent a printer or a file on a disk.

Example:

```
10 - -
20 OPEN "PR:" AS FILE 2 ! Open the printer
30 - -
40 PRINT #2, "Hello"
50 CLOSE 2 ! Close the printer
60 END
```

Note: If no device is specified in the PRINT statement (i.e., omit #2 in statement 40 above) then CON: is assumed. CON: stands for console (keyboard and screen). Channel #0 is always opened to the console and cannot be closed.

## 2.12 ERROR HANDLING

Certain errors can be detected by Monroe BASIC when it executes a program. These errors can, for instance, be computational errors (such as division by 0) or input/output errors (reading an end-of-file code as the input to an INPUT statement). Normally, the occurrence of any of these errors will cause termination of program execution and the printing of a diagnostic message. The file BASICERR/ASC must be on the system volume; otherwise, just the error number will be printed.

Some applications may require that program execution continues after an error has occurred. To accomplish this, the user can include an ON ERROR GOTO <line number> statement in the program. The program will then jump to a subroutine, which begins at the specified line number. The subroutine can contain an error handling routine, which will analyze the error in question.

The ON ERROR GOTO statement should be placed before all the executable statements with which the error handling routine deals.

When an error occurs in a program, Monroe BASIC checks to see if the program has executed an ON ERROR GOTO statement. If not, a message is printed at the screen and the program execution is terminated. If an ON ERROR GOTO statement has been executed, the program execution will continue at the line number specified by that statement. The subroutine at that line number can test the function ERRCODE to find out precisely what error has occurred and decide what action is to be taken.

If there are portions of the program in which any errors detected are to be processed by the system and not by the subroutines of the program, the error subroutine can be disabled by executing the following statement:

```
ON ERROR GOTO
```

The computer will then attend to all errors as it would do if no ON ERROR GOTO <line number> had ever been executed.

The error handling routine is terminated by a RESUME statement. The function of RESUME resembles the one of the RETURN statement at the end of an ordinary subroutine; the program resumes at the beginning of the statement that caused the error. If the program execution should continue at another line number, the line number desired should be given in the RESUME statement.

Example of error handling:

```
LIST 10-120¶
10  ON ERROR GOTO 100 !At erroneous input go to line 100
20  INPUT "AGE, WEIGHT " A,W
30  ON ERROR GOTO !Disable the error handler
40  STOP
100  PRINT !Error handler
110  PRINT " Erroneous input! "
120  RESUME !Jump to line 20
BASIC
```

## 2.13 FUNCTION KEYS

The console has eight function keys labelled F1/F9 through F8/F16.

A programmer can assign various functions to the function keys, e.g. cursor movements, write data, read data, update data or a jump to a program module.

The function keys can produce 32 different ASCII values as shown in Table 2-1.

Table 2-1. Function Key ASCII Values

| Key | Normal | Shift | CTRL | Shift+CTRL |
|---|---|---|---|---|
| F1/F9 | 128 | 136 | 144 | 152 |
| F2/F10 | 129 | 137 | 145 | 153 |
| F3/F11 | 130 | 138 | 146 | 154 |
| F4/F12 | 131 | 139 | 147 | 155 |
| F5/F13 | 132 | 140 | 148 | 156 |
| F6/F14 | 133 | 141 | 149 | 157 |
| F7/F15 | 134 | 142 | 150 | 158 |
| F8/F16 | 135 | 143 | 151 | 159 |
| RETURN | 13 | | | |
| RUN | 208 | | | |
| LOAD | 209 | | | |
| CONTINUE | 210 | | | |
| HOME | 199 | | | |
| ↑ | 197 | | | |
| ↓ | 198 | | | |

The function keys will act as data terminators in all operating modes, unless keyboard input is being analyzed by the user's program on a single byte basis (e.g., by use of the GET <string variable> statement in Monroe BASIC). In addition, the RETURN, RUN, LOAD, CONTINUE, HOME, cursor up and cursor down keys act as terminators.

To determine which terminating key has been depressed, use the SYS(7) function in conjunction with the values in Table 2-1.

The use of one of these alternate terminating keys will have no effect on treatment of the data entry.

Example: When a specific function key is depressed, a corresponding subroutine can be called as shown below.

```
LIST¶
.
.
.
 90 ; "TERMINATE DATA INPUT BY F1, F2, OR F3"
100 INPUT "DATA?" A$
110 A=SYS(7)
120 ON A-127 GOSUB 1000, 2000, 3000 ¶
.
.
.
```

Here control is transferred to statement 1000, 2000 or 3000 depending on whether the input data was terminated by F1, F2, or F3 respectively.

For a complete list of all the possible ASCII codes capable of being generated by the 8800 Series Keyboard see Appendix A.

## 2.14 CHANGING THE SYSTEM DISK

Once the operating system and Monroe BASIC have been loaded from the system volume, that volume need no longer be mounted. It is sometimes necessary or desirable to remove the system disk in order to mount another volume for the purpose of loading or saving a program on that volume; this is particularly relevant to the single disk EC8800.

The ability to change the system disk allows you to keep system programs and utilities on a separate disk from your BASIC programs and data files. This can be an advantage when you have a large number of BASIC programs or data files since it will reduce the total number of diskettes required to contain them. It also allows the maximum amount of on-line disk space for application programs and data.

Your data disk should contain any system utility programs that you might want to use while your data diskette is mounted, e.g., CMD$LIB/T is required if you will want to use the LIB command. If you want BASIC to print textual error messages instead of error numbers when a BASIC error occurs, the file BASICERR/A should be copied to your data disk.

The manual OPEN and CLOSE commands for changing the system disk can be implemented by a BASIC program directly through SVCs; see Section 14 of this manual and the 8800 Series Monroe Operating System Programmer's Reference Manual for additional information.

### Single Disk System

The following procedure should be followed when switching disks on a single disk system; the VOLUME command need be entered only the first time you switch disks. The file CMD$VOLUME/T must be on the disk you booted from.

```
BASIC
CLOSE¶
BASIC
PAUSE¶
```

```
00.00.00 Paused
-VOLUME -:¶(only the first time you switch disks)
-CLOSE FPY0:¶
          (now switch the disks)
-OPEN FPY0:¶
FPY0 (new volume name)
-         (now type the CONT key)
BASIC
```

Multiple Disk System

The following is the recommended procedure for changing the system volume; it requires that a copy of the file CMD$VOLUME/T be on each new volume to be mounted. This procedure can be repeated to switch to other data disks or back to the original system disk.

```
BASIC
CLOSE¶
BASIC
PAUSE¶

00.00.00 Paused
-CLOSE FPY0:¶
          (now switch the disks)
-OPEN FPY0:¶
FPY0 (new volume name)
--:VOLUME <new volume name>¶ (type the name just output above)
-         (now type the CONT key)
BASIC
```

Note that the first dash shown preceding the VOLUME command is the console monitor prompt; you must type the second dash. If a "Load-error 75" results when the VOlUME command is issued, the file CMD$VOLUME is not on the disk to be mounted; you must remount and reOPEN the system disk and use the alternate sequence which follows.

If the disk to be mounted does not contain the file CMD$VOLUME, the order of the CLOSE, OPEN and VOLUME commands above must be changed as follows; this sequence requires that you know the name of the new volume to be mounted:

```
-VOLUME <new volume name>¶
-CLOSE FPY0:¶
                (now switch the disks)
-OPEN FPY0:¶
FPY0 (new volume name)
```

If the volume name printed out by the OPEN command does not match the volume name you entered in the VOLUME command or if you subsequently want to switch to another data disk, you must remount and reOPEN the system disk and then repeat one of the above sequences.

The above procedures are the safest in the sense that that default system volume is always a single known volume. The "-" character can be supplied for the new volume name in the VOLUME command, in which case the VOLUME command need only be issued once and the file CMD$VOLUME/T need only be present on the volume you boot from. If this is done, the system will search all mounted volumes to find an existing file. However, when creating a new file (or replacing an old one), the volume name should be explicitly specified to ensure that the file will be created on the volume you intend. In addition, when the same filename appears on more than one volume, the volume name should be explicitly specified to ensure that the correct file will be referenced.

# SECTION 3
# FORMING EXPRESSIONS

# SECTION 3
# FORMING EXPRESSIONS

Expressions are a fundamental building block used in many Monroe BASIC statements. The primary elements of expressions are constants, variables, arrays and functions. These elements are then combined using arithmetic, relational and/or logical operators, to form expressions. This and succeeding sections will define these terms within the context of Monroe BASIC.

## 3.1 ARITHMETIC EXPRESSIONS

An arithmetic expression has an arithmetic value which is either floating point or integer. Mixed expressions (i.e., both floating point and integer) yield a floating point value. The following mathematical operators can be used in arithmetic expressions:

| Operator | Function |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| ^ or ** | Exponentiation |
| - (unary) | Subtraction or negation |

No two mathematical operators may appear in sequence and no operator is ever assumed (e.g., A++B and (A+2) (B-3) are not valid).

Examples of Arithmetic Expressions:

```
4.123
3 + A
A% +50
B * (C**3 + 1.5)
PI *R**2
```

## 3.2 RELATIONAL EXPRESSIONS

A relational expression yields a truth value that reflects the result of comparing two values. Symbolically it can be defined as:

<expression><relational operator><expression>

The expression can be either arithmetic terms or string terms but not both in a single relational expression.

The relational symbols Monroe BASIC allows are:

| Mathematical Symbol | Monroe BASIC Symbol | Example | Meaning |
|---|---|---|---|
| = | = | A=B | A is set equal to B |
| < | < | A<B | A is less than B |
| > | > | A>B | A is greater than B |
| ≤ | <= | A<=B | A is less than or equal to B |
| ≥ | >= | A>=B | A is greater than or equal to B |
| ≠ | <> | A<>B | A is not equal to B |

Examples:

```
X>Y
NUM8<=0
A=B
```

Examples of string relational symbols are shown in Section 5.

## 3.3 LOGICAL EXPRESSIONS

A logical expression yields a truth value that reflects the existence or nonexistence of a particular condition.

A logical expression is one of the following:

1. An integer expression (FALSE if 0, TRUE if <> 0).
2. A set of relational expressions, connected by logical operators.
3. A set of integer expressions, or logical expressions, or both, connected by logical operators.

Logical operators are used in IF - THEN and such statements where some condition is used to determine subsequent operations within user program.

The logical operators are as follows (where A and B are relational expressions):

| Monroe BASIC Operator | Example | Meaning |
|---|---|---|
| NOT | NOT A | The logical negative of A. IF A is true, NOT A is false. |
| OR | A OR B | A OR B has the value true if either or both A or B are true and has the value false only if both A and B are false. |
| XOR | A XOR B | The logical exclusive OR of A and B. A XOR B is true if either A or B is true but not both, and false otherwise. |
| IMP | A IMP B | The logical implication of A and B. A IMP B is false if and only if A is true and B is false; otherwise the value is true. |
| EQV | A EQV B | A is logically equivalent to B. A EQV B has the value true if A and B are both true or both false, and has the value false otherwise. |
| AND | A AND B | The logical product of A and A. A AND B has the value true only if A and B are both true and has the value false if A or B is false. |

## 3.4 DATA TYPES

Three types of data are supported by Monroe BASIC: Floating Point, Integer and String Values.

### Floating Point Values

Floating point values range from:

$2.93874 \times 10^{-39}$ through $1.70141 \times 10^{38}$ - single precision

$2.938735877055719 \times 10^{-39}$ through $1.701411834604692 \times 10^{38}$ - double precision

All floating point variables and expressions are calculated to single or double precision. Mixing of types is not possible. The default is single precision.

Single precision allows for six digits of significance and double precision allows for sixteen digits. Numbers are internally rounded, using 5/4 round method to fit the appropriate precision. Numbers may be entered and displayed in three formats:

1. Whole - 153
2. Fractional - 34.52
3. Scientific Notation (E-format) - 136E-2

### Integer Values

The range of integer numbers is:

-32768 through 32767

### String Values

A string can contain any number of characters.

Note: Strings used in string arithmetic have a maximum size of 126 characters including the sign and the decimal point.

## 3.5 CONSTANTS

Numeric constants retain a constant value throughout a program. They can be positive or negative. Numeric constants can be written using decimal notation as follows:

```
+3
-4.567
12345.6
-.0001
```

The example constants would normally be stored as floating point, since they have no % suffix.

The use of an explicit decimal point or percent sign is recommended in all numeric constants to avoid unnecessary data conversions and to improve documentation.

## 3.6 VARIABLES

A variable is a data item whose value can be changed during program execution. A numeric variable is denoted by a fixed variable name.

Two modes dictate the length of a variable name: EXTEND and NO EXTEND.

In EXTEND mode variable names of up to 32 characters are permitted, but spaces are required to delineate names and functions unless the adjoining characters are a line number or arithmetic operator. In NO EXTEND mode variable names of one letter and an optional digit are allowed but spaces are unnecessary. The default is NO EXTEND mode. The following are the letters and digits which can be used to form variable names: A,B,...,Z and 0,1,...,9.

A name can also have an FN prefix (denoting a function name), a % suffix (denoting an integer), a . suffix (denoting floating point), a $ suffix (denoting a string), or a subscript suffix that consists of a set of subscripts enclosed in parentheses.

A string expression is a value that consists of a sequence of characters, each character occupying a byte. A string expression can be expressed either as a sequence of characters enclosed in quotation marks or as a variable by a variable name with a $ suffix.

For efficiency considerations, mixing of data types in a statement should be avoided if possible. Use integers whenever possible.

The same name in combination with various prefixes and suffixes can appear in the same program and generate mutually independent variables. For example, the name A refers to a floating point variable A. The name A can be used as follows:

| | |
|---|---|
| A | floating point variable A |
| A% | integer variable A% |
| A$ | string variable A$ |
| A(d) | floating point array A with dimension subscript d |
| A%(d) | integer array A% with dimension subscript d |
| A$(d) | string array A$ with dimension subscript d |
| FNA | floating point function A |
| FNA% | integer function A% |
| FNA$ | string function A$ |

In the EXTEND mode a name can be used as follows:

| | |
|---|---|
| SECANT | floating point variable SECANT |
| SECANT% | integer variable SECANT% |
| SECANT$ | string variable SECANT$ |
| SECANT(d) | floating point array SECANT with subscript d |
| SECANT%(d) | integer array SECANT% with subscript d |
| SECANT$(d) | string array SECANT$ with subscript d |
| FNSECANT | floating point function SECANT |
| FNSECANT% | integer function SECANT% |
| FNSECANT$ | string function SECANT$ |

Variables are assigned values by LET, INPUT and READ among other statements. Variables are set to zero before program execution.

It is necessary to assign a value to a variable only when an initial value other than zero is required. To ensure that later changes or additions will not cause problems it is good programming practice to always initialize all variables to zero.

## 3.7 SUBSCRIPTED VARIABLES (ARRAY) AND THE DIM STATEMENT

In addition to the simple variables the use of subscripted variables (arrays) is allowed. Subscripted variables provide the programmer with additional computing capabilities for dealing with lists, tables, matrices, or any set of related variables. Variables are allowed one (vector) or two or more (matrix) subscripts.

The name of a subscripted variable is any acceptable variable name followed by one or two integers enclosed in parentheses. For example, a list might be described as A(I) where I goes from 0 to 5 as follows: A(0), A(1), A(2), A(3), A(4), A(5).

This allows the programmer to reference each of six elements in the list, which can be considered a 1-dimensional algebraic vector as follows:

A(0)
A(1)
A(2)
A(3)
A(4)
A(5)

A 2-dimensional matrix B(I,J) can be defined in a similar manner. It is graphically illustrated below:

| | | | | | |
|---|---|---|---|---|---|
| B(0,0) | B(0,1) | B(0,2) | B(0,3) | ... | B(0,J) |
| B(1,0) | B(1,1) | B(1,2) | B(1,3) | ... | B(1,J) |
| B(2,0) | B(2,1) | B(2,2) | B(2,3) | ... | B(2,J) |
| B(3,0) | B(3,1) | B(3,2) | B(3,3) | ... | B(3,J) |
| . | . | . | . | | . |
| . | . | . | . | | . |
| . | . | . | . | | . |
| B(I,0) | B(I,1) | B(I,2) | B(I,3) | ... | B(I,J) |

Subscripts used with subscripted variables can only be integer values. Subscripts are truncated to integers if they are of floating type.

A DIM dimension statement is used to define the maximum number of elements in an array. DIM statements are executable and can decrease, but not increase subscript limits established by previously executed DIMs.

Arrays may start with subscript 0 or 1. An array dimensioned A (5), will have 5 elements if option base 1 is specified or 6 elements if option base 0 is specified. The default is option base 0. If an option base is specified, it must be declared before any array is dimensioned or used.

If a subscripted variable is used without a DIM statement, it is assumed to be dimensioned to length 9 or 10 in each dimension (that is, having 10 or 11 elements in each dimension, 1 through 10 or 0 through 10 respectively). DIM statements are usually grouped together among the first lines of a program.

The first element of every matrix is automatically assumed to have a subscript of (0,0), if OPTION BASE 1 is not specified.

Example:
```
OPTION BASE 0
 10   REM - MATRIX CHECK PROGRAM¶
 20   DIM A(4,8)¶
 30   FOR I=0 TO 4¶
 40   LET A(I,0)=I¶
 50   FOR J=0 TO 8¶
 60   LET A(0,J)=J¶
 70   PRINT A(I,J);¶
 80   NEXT J¶
 90   PRINT¶
100   NEXT I¶
999   END¶
```

```
RUN¶
 0   1   2   3   4   5   6   7   8
 1   0   0   0   0   0   0   0   0
 2   0   0   0   0   0   0   0   0
 3   0   0   0   0   0   0   0   0
 4   0   0   0   0   0   0   0   0
```

Example: OPTION BASE 1

```
 10    REM - MATRIX CHECK PROGRAM¶
 15    OPTION BASE 1¶
 20    DIM A(4,8)¶
 30    FOR I=1 TO 4¶
 40    LET A(I,1)=I¶
 50    FOR J=1 TO 8¶
 60    LET A(1,J)=J¶
 70    PRINT A(I,J);¶
 80    NEXT J¶
 90    PRINT¶
100    NEXT I¶
999    END¶

RUN¶
 1   2   3   4   5   6   7   8
 2   0   0   0   0   0   0   0
 3   0   0   0   0   0   0   0
 4   0   0   0   0   0   0   0
BASIC
```

Notice that a matrix element, like a simple variable, has a value of 0 until it is assigned a value.

# SECTION 4
# ARITHMETIC OPERATIONS

# SECTION 4
# ARITHMETIC OPERATIONS

Normally, all numeric values (variables and constants) specified in a Monroe BASIC program are stored internally as floating point numbers. If the numbers to be dealt with in a program are integers, significant economies in storage space can be achieved by use of the integer data type. Integer arithmetic is also faster than floating point arithmetic. This section discusses integer and floating point operations within the context of Monroe BASIC.

A constant, variable or function can be specified as an integer by ending its name with the % character.

Example:

```
A%   FNX% (y)
-8%  Z3%
```

Unless the INTEGER statement has been entered, the user always has to terminate a value with the %-character to indicate where an integer is to be generated. Otherwise, a floating point value is produced.

When raising to an integer power, the power value should be indicated explicitly as an integer.

## 4.1 MATHEMATICAL OPERATIONS

When more than one operation is to be performed in a single formula, rules are observed as to the precedence of the operators. The arithmetic operations are performed in the following sequence. Operation described in item 1 has precedence.

1. Any formula within parentheses is evaluated first. Then the parenthesized quantity is used in further computations. Where parentheses are nested the innermost parenthetical quantity is calculated first. For example, (A+(B*(C**3))) is evaluated as follows:

   Step 1 - (C**3), Step 2 - (B*(C**3)), and Step 3 - (A+(B*(C**3))).

2. In absence of parentheses the following precedence is performed:

   a. Intrinsic or user-defined functions
   b. Exponentiation (**)
   c. Unary, minus (-), that is, a negative number or variable
   d. Multiplication and division (* and /)
   e. Addition and subtraction (+ and -)
   f. Relational operators (=, <>, >=, <, <=, >)
   g. NOT
   h. AND
   i. OR and XOR
   j. IMP
   k. EQV

   Thus, for example, -A**B with a unary minus, is a legal expression and is the same as -(A**B). This implies that -2**3 evaluates as -8. The term A**-B is not allowed; however, A**(-B) is allowed.

3. In absence of parentheses, operations on the same level are performed left to right, in the order the formula is written.

## 4.2 INTEGER ARITHMETIC

Addition or subtraction with integer variables is performed in modulo 2**16. The number is -32,768 to +32,767 and can be regarded as a continuous circle with -32,768 following +32,767. The result of an integer multiplication or exponentiation operation must be contained in the above range or an error message will be generated.

Integer division forces truncation of any remainder. Note that the function MOD makes the remainder available.

Example:

3%/4% = 0 and 283%/100% = 2.

When an arithmetic operation is performed between integer and floating point operands, the result is floating point. When the result of an operation is assigned to a variable, it is converted to the mode of the variable to which it is assigned. When a floating point value is assigned to an integer variable, the value is rounded up to the nearest integer if the fractional part is greater than or equal to 0.5.

Example:

```
10   LET B% = Z% + 3/X
```

The result is rounded to give B% an integer value.

## 4.3 INPUT/OUTPUT WITH INTEGERS AND FLOATING POINT

Input and output of integer variables is performed in exactly the same manner as the corresponding operations on floating point variables.

Any number which can be represented by up to six significant digits in single precision mode (or 16 digits in DOUBLE precision mode) is printed without using the exponential form.

Any floating point variable that has an integer value is automatically printed as an integer but is internally still a floating point number.

If more than six digits (single precision) or sixteen digits (double precision) are generated during any computation using floating point numbers, the result is automatically printed in E format:

[-] x Ey

where: -= Sign of the number, if number is negative

x = A maximum of six digits for single precision and sixteen digits for double precision

E = Represents the expression "times 10 to the power of"

y = An exponent in the range (-38 through +38)

Examples:

5E-06 = $5 \times 10^{-6}$ = .000005

-125E+4 = $-125 \times 10^{4}$ = -1250000

Input allows all the formats used in output.

## 4.4 USER-DEFINED FUNCTIONS

An integer function is defined to be of integer type by including the "%" suffix following the function name.

Example:

```
10    DEF FNA%(X%) = X% * (Z% + X%)
```

A floating point function could be written as:

Example:

```
10    DEF FNV (X%)=X%*(Z+X%)
```

## 4.5 USE OF INTEGERS AS LOGICAL VARIABLES

Integer variables or integer valued expressions can be used within IF statements in any place that a logical expression can appear. Any non-zero value is defined to be true and an integer value of 0% corresponds to the logical value false. The logical operators (AND, OR, NOT, XOR, IMP, EQV) operate on logical (or integer) data in a bit-wise manner.

Note: Logical values generated by Monroe BASIC always have the values -1% (true) and 0% (false).

## 4.6 LOGICAL OPERATIONS ON INTEGER DATA

Monroe BASIC permits a user program to combine integer variables or integer valued expressions using a logical operator to give a bit-wise result.

For the purpose of logical operations the truth tables following are valid. A is the condition of one bit in one integer value and B is the condition of the bit in the corresponding bit position of another integer value.

The truth tables are as follows:

| A | B | A AND B | A OR B | A XOR B | A EQV B | A IMP B | NOT A |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

The result of a logical operation is an integer value generated by combining the corresponding bits of two integer values according to the rules shown above.

The result of any logical operation can be assigned to an integer or a floating point variable.

Example:
```
10   REM BIT VALUES: 13 = 00001101, 14 = 00001110¶
20   REM ** 13 or 14 = 00001111 = 15¶
30   A% = 13% or 14%¶
40   PRINT A%¶
RUN¶
15
BASIC
```

Variables and valued expressions can be operated on by AND, OR, XOR, EOV, IMP and NOT to give a bit-wise integer result. If logical operations are done on floating point variables or floating point valued expressions, conversion to integer format is done before the execution of the logical operation.

Example:
```
    100 IF A% AND 1% THEN ...
is the same as:
    100 IF A% AND 1.6 THEN ...
```

# SECTION 5
# CHARACTER STRINGS

# SECTION 5
# CHARACTER STRINGS

Besides the manipulation of numerical information Monroe BASIC also processes information in the form of character strings. A character string is a sequence of characters. This section defines string elements within the context of Monroe BASIC.

## 5.1 STRING CONSTANTS

Character string constants are allowed analogous to numerical constants. Character string constants are delimited by either single (') or double (") quotes. If the delimiting character occurs twice in a string sequence it is considered as part of the string constant as a single occurence.

For example, the string value BOB'S can be expressed in two ways: "BOB''S" or 'BOB''S'.

Examples:

```
10  A1$ = "CHARLES"
20  IF A$ = "GOOD" GOTO 40
30  B$ = 'DON''T'              (has the value DON'T)
```

## 5.2 STRING VARIABLES

Any legal name followed by a dollar sign ($) character is a legal name for string variable.

Examples:

A$,B4$ are simple string variables.
B$(8),H5$(N,0),J$(K) are subscripted string variables.
AMOUNT$(4) - (EXTEND MODE ONLY)

Note: The same name, without the $, denotes a numeric variable which can be used in the same program.

Example:

A,A$ and % are allowed in the same program.

## 5.3 SUBSCRIPTED STRING VARIABLES

The DIM-statement is used to define string lists and string matrices.

Examples:

```
DIM W$(2,4)=8 !STRING LENGTH 8 maximum subscript values
2 and 4
DIM R5$(9,9) !STRING LENGTH UP TO 80; maximum subscript
values 9 and 9
DIM NAME$(7,6,3,2)=10 !STRING LENGTH 10; four-
dimensional matrix with maximum subscript values 7,6,
3, and 2
```

## 5.4 STRING SIZE

The maximum length of a non-dimensioned string variable is automatically set to the current length the first time the string is assigned a non-null value (<>''). _

If less than 80 characters are assigned to the variable, then a default maximum length of 80 characters is assigned.

Each string, both scalar and each array element, has two lengths:

1. The maximum length is the number of bytes in memory allocated to the string.

2. Current length is the number of bytes currently in use. Current length may vary between zero and the maximum length. The current length is the only visible length; this length may be examined by the function LEN, etc.

Both lengths are initialized to zero as a program is started. They are modified when the string is dimensioned or assigned. If a string is assigned a null value (='') the current length will be set to zero. No further action is taken.

## 5.5 STRING FUNCTIONS

Monroe BASIC provides various functions for use with character strings. These functions permit the program to:

- perform arithmetic operations with numeric strings
- concatenate two strings
- access part of a string
- determine the number of characters in a string
- generate a character string corresponding to a given number or vice versa
- search for a substring within a large string, etc.

Section 10 discusses each string function in detail.

## 5.6 STRING ARITHMETIC

The string arithmetic features functions that treat numeric strings in arithmetic operands. This is a way to perform calculations with greater precision. Numeric string variable names must be suffixed with a dollar sign ($) character. Numeric string constants must be bounded by quotation marks (") or apostrophes (').

The maximum size of a string arithmetic operand is 125 characters including the sign and the decimal point.

## 5.7 STRING INPUT

The READ, DATA and INPUT statements can also be used to assign data to string variables in a program.

Example:

```
10  INPUT "YOUR ADDRESS?";A$,
20  INPUT "YOUR NAME?";B$
is the same as
10  PRINT "YOUR ADDRESS";
20  INPUT A$,
30  PRINT "YOUR NAME";
40  INPUT B$
```

INPUT LINE is useful for string input because it accepts leading, trailing and embedded blanks, commas, etc. It accepts only one line from the keyboard and appends carriage return and line feed characters to the string.

Example:

```
10  INPUT LINE D$
```

Example:

```
10  READ A, B, C$, D
20  DATA 17, 14, 61, 4
```

This results in the following assignments:

```
A  = 17
B  = 14
C$ = character string "61"
D  = 4
```

The INPUT statement is used to input character strings exactly as though accepting numeric values.

## 5.8 STRING OUTPUT

Only those characters that are within quotes are printed when character string constants are included in PRINT statements. The delimiters are not printed:

Example:

```
10 PRINT "ALL IS OKAY"¶
RUN¶
ALL IS OKAY
BASIC
```

Strings can also be output to disk files or an output device (e.g., PR:). Formatted string output is possible with the PRINT USING statement.

## 5.9 RELATIONAL OPERATORS

The relational operators, when applied to string operands, indicate alphabetic sequence.

Example:

```
15  IF  A$(I)<A$(I+1)   GOTO 115
```

When line 15 is executed the following occurs: A$(I) and A$(I+1) are compared; if A$(I) occurs earlier in alphabetical order than A$(I+1), execution continues at line 115.

The chart below contains a list of the relational operators and their string interpretations.

| Operator | Example | Meaning |
|---|---|---|
| = | A$=B$ | The strings A$ and B$ are equivalent. |
| < | A$<B$ | The string A$ occurs before B$ in collating sequence. |
| <= | A$<=B$ | The string A$ is equivalent to or occurs before B$ in collating sequence. |
| > | A$>B$ | The string A$ occurs after B$ in collating sequence. |
| >= | A$>=B$ | The string A$ is equivalent to or occurs after B$ in collating sequence. |
| <> | A$<>B$ | The strings A$ and B$ are not equivalent. |

When two strings of unequal length are compared, the shorter string (of length n) is compared with the first n characters of the longer string. If this comparison is not equal, that inequality serves as the result of the original comparison. If the first n characters of the strings are the same, the longer string is greater than the shorter string; trailing blanks are significant in string comparisons.

A null string (of length zero) is less than any string of length greater than zero.

# SECTION 6
# CONTROL COMMANDS

SECTION 6

CONTROL COMMANDS

## 6.1 INTRODUCTION

It is possible to communicate with the Monroe BASIC interpreter entering direct commands from the keyboard. Also, certain other statements can be directly executed when they are given without statement numbers.

Commands have the effect of causing Monroe BASIC to take immediate action. A Monroe BASIC language program, by contrast, is first entered into the memory and then executed later when the RUN command is given.

When Monroe BASIC is ready to receive a command, the word BASIC is displayed on the screen. Commands should be typed without any line numbers.

After a command has been executed, the user will either be prompted for more information, or the BASIC prompt will again be displayed. This indicates that Monroe BASIC is ready for more input, either another command or program statements.

Example:

```
          |
          |
          100
          110
          <command>
          BASIC
          |
          |
          <command>
          |
          |
          BASIC
```

Commands control the editing and execution of programs and allow files to be manipulated. Each command is identified by a keyword at the start of the line. Keywords are shown in upper-case letters. All characters of the keyword are mandatory.

Table 6-1 lists the Monroe BASIC control commands described in this section along with a short description for each.

Table 6-1. Monroe BASIC Control Commands

| Command | Description |
|---|---|
| AUTO | Generates line numbers automatically. |
| CLEAR | Clears all variables and closes all files. |
| CONT (or CON) | After CTRL C operation this command restarts the program at the line at which it stopped. |
| ED | Gives program editing facilities. |
| ERASE | Deletes blocks of lines from a Monroe BASIC program. |
| LIST | Outputs a program to a specified device. |
| LOAD | Loads the program requested into memory from a specified device. |
| MERGE | Inputs lines from a program on disk to the current program. |
| NEW | Deletes the current program and resets all variables. |
| RENUMBER (or REN) | Changes the line numbering. |
| RUN | Executes a Monroe BASIC program. |
| SAVE | Stores the current program on a disk. |
| UNSAVE | Deletes a non-protected program from the disk. |

The following sections describe the function, type, format, arguments and use of each of the above commands. Examples are included to show how the command can be used. Errors may occur when using a command incorrectly or syntaxically wrong. A complete list of error messages is shown in Appendix B.

## 6.2 AUTO COMMAND

**Function:** Generates line numbers automatically after each carriage return.

**Mode:** Direct

**Format:**

1. AUTO
2. AUTO <lineno.1>
3. AUTO <line no.1>,<interval>

**Arguments:** Line no.1 specifies the start line and interval specifies the step value.

Both line no.1 and interval are optional. If no arguments are given, then the line numbering starts with the next whole 10th number (i.e., 10, 20, 30, etc.) after the existing line numbers. The step is set to 10 if the new interval is not included.

**Use:** AUTO facilitates freedom from line numbering. It is continuously available during the programming work. Automatic line generation stops when the carriage return is entered as first character on a line. If an explicit line number is entered, BASIC uses it instead of the line number specified by auto mode, and it will reprompt with the same line number. If a line entered causes an error message, automatic line numbering is stopped and the line can be edited. The line numbering can be started by a new AUTO command.

**Examples:**

Ex. 1

```
AUTO 10,5
```

The first line number will be 10 and the line number will be incremented by 5 for each line.

```
AUTO 10,5¶
10 LET A=1¶
15 - - -¶
20 - - -¶
25 - - -¶
```

Ex. 2

```
AUTO¶
10 INPUT "CYLINDER HEIGHT = ", H¶
20 INPUT "CYLINDER RADIUS = ", R¶
30 PRINT "CYLINDER VOLUME -": 2*PI*R*H¶
40 END¶
50 ¶
BASIC
NEW
AUTO 50¶
50 INPUT "CYLINDER HEIGHT = ", H¶
60 INPUT "CYLINDER RADIUS = ", R¶
70 PRINT "CYLINDER VOLUME -",2*PI*R*H¶
80 END¶
90 ¶
BASIC
NEW
AUTO 100,5
100 INPUT "CYLINDER HEIGHT = ",H¶
105 INPUT "CYLINDER RADIUS = ",R¶
110 PRINT "CYLINDER VOLUME -",2*PI*R*H¶
115 END¶
120 ¶
BASIC
```

## 6.3 CLEAR COMMAND

**Function:** Clears all variables and closes all open files.

**Mode:** Direct

**Format:** CLEAR

**Action:** CLEAR does not affect the existing program which is still left in memory.

**Example:**

```
10 A%=1234%
20 END
RUN
BASIC
; A%
  1234
BASIC
CLEAR
BASIC
; A%
0
BASIC
A%=4567%
BASIC
; A%
  4567
BASIC
```

## 6.4 CONTINUE COMMAND

**Function:** Continues program execution from where it was stopped by either CTRL C entered twice or a STOP statement.

**Mode:** Direct

**Format:**
1. CON
2. <CONT key>

**Action:** Execution of "CON" causes the program to restart at the line at which it stopped.

**Use:** A variable may be displayed and changed using a direct mode statement before "CON" is used. If the program is edited or an "END" statement caused the program to be terminated, the "CON" statement will cause an error and should not be used.

**Example:**

```
10 FOR I = 1 to 10000¶
20 ;I;¶
30 NEXT I¶
40 END¶

RUN¶
1 2 3 4 5 ...

CTRL C (enter twice)
CON

.............  (Continue Printing)

BASIC
```

## 6.5 EDIT COMMAND

**Function:** Allows a previously entered program line to be edited.

**Mode:** Direct

**Format:** ED <line no.>

**Argument:** Line no. is the line to be corrected.

**Use:** Once the command is entered, the line specified will be displayed. The cursor is positioned after the last character on the line. Refer to Section 2.6, "Immediate Corrections" for the list of editing keys which can be used at this point.

**Note:** When editing a program line with the editing keys, entering a DELETE LINE prior to a return will negate any changes that have been made. Note that when an Edit command is issued, an implicit CLEAR Command is performed.

The line number of a line can also be edited to copy a line to a different portion of the program; the original line will remain, but can be deleted if necessary.

**Examples:**

```
LIST¶
10  A$="1.4726"
20  B$="7.75"
30   ;ADD$(A$,B$,4)
BASIC
```

Examples:

1. To change 1.4726 to 1.46 in line 10 do the following:
   a. ED 10
   b. Depress cursor left (←) three times
   c. Depress DELETE CHAR twice
   d. Depress RETURN key

2. To insert 423 before 75 in line 20, do the following:
   a. ED 20
   b. Depress cursor left (←) twice
   c. Depress INSERT CHAR lock
   d. Enter 423
   e. Depress RETURN key

```
LIST 10-20¶
10 A$ = "1.46"
20 B$ = "7.42375"
BASIC
```

## 6.6 ERASE COMMAND

**Function:** Deletes blocks of lines from the current program.

**Mode:** Direct

**Format:** ERASE <line no(s)>

**Arguments:** Line no(s) can be the single line number or a range of line numbers to be listed. A single line number can have a "-" appended to or before it (e.g., 10-, -10) to designate all lines up to 10 or from 10 to the end of the program are to be listed, respectively.

**Use:** All lines between line no. 1 through line no. 2 are removed.

**Examples:**

```
ERASE 20-200 ! ERASE LINES 20 UP TO AND INCLUDING
200
ERASE -100 ! ERASE ALL LINES UP TO AND INCLUDING
LINE 100
ERASE 50- ! ERASE FROM LINE 50 TO END OF PROGRAM
```

## 6.7 LIST COMMAND

**Function:** Lists all or part of the current program to the console, printer or disk.

**Mode:** Direct

**Format:**

1. LIST [fd]
2. LIST [line no(s)]
3. LIST <fd>[,line no(s)]

**Arguments:** fd is the file descriptor as previously defined in Section 1.3.

Line no(s) can be the single line number or range of line numbers to be listed. A single line number can have a "-" appended to or before it (e.g., -10, 10-) to designate all lines up to 10 or from 10 to the end of the program are to be listed, respectively.

**Use:**

1. LIST VOL:XYZ

   Saves a program in an uncompressed way on the disk with volume VOL under specified file name XYZ and type "AB" (ASCBAS). Note that only a file saved with LIST rather than SAVE can be edited or listed by a utility outside of Monroe BASIC.

2. LIST

   The entire program is listed.

3. LIST 100

   Line 100 is displayed on the screen.

4. LIST 100-1000

   All lines between 100 and 1000 inclusive are displayed on the screen.

5. LIST PR:

   The entire program is output on the printer.

6. LIST -1500

   All lines up to 1500 are listed.

7. LIST 2000-

   All lines from 2000 through the last line are listed.

Note: If a large program is listed, the listing will stop after one full page has been displayed. The next line will be displayed when you press the space bar. A long listing may be stoped by pressing CTRL C, RETURN or entering any Monroe BASIC command/ statement. If the TAB key is entered after the listing has stopped, an implicit EDIT command is issued for the last line number appearing on the screen.

Examples:

```
LIST ACCT:PAYROLL ! SAVE FILE PAYROLL ON DISK ACCT
LIST ! LISTS THE ENTIRE PROGRAM ON THE SCREEN
LIST 100 ! LISTS LINE 100
LIST 100 - 500 ! LISTS LINES 100 TO 500
LIST PR: ! LISTS THE ENTIRE PROGRAM ON PRINTER
LIST PR:, 100-200 ! LISTS LINES 100-200 ON PRINTER
```

## 6.8 LOAD COMMAND

Function: Loads a Monroe BASIC program from external storage into working storage.

Mode: Direct

Format:
1. LOAD <fd>
2. <fd><LOAD key>

Arguments: fd is the file descriptor as previously defined in Section 1.3.

Note that when the file type is omitted, the computer will look for type BAC (compressed form) first and then ASCBAS (uncompressed form).

If type "AB" (ASCBAS) is specified only ASCBAS will be searched for in the directory.

Use: Loads the specified file after having cleared the working memory.

All open files are closed, the program area and buffers are reset. All variables are erased.

Examples: Ex. 1

```
LOAD TEST:ABC200
```

Filename ABC200 on volume TEST is read, not to the END statement, but to the EOF (end of file).

Example: Ex. 2

```
LOAD MAST/AB
```

Filename MAST in uncompressed format is to be loaded into working storage.

## 6.9 MERGE COMMAND

**Function:** Merges lines from a file into the current program.

**Mode:** Direct

**Format:** MERGE <fd>

**Arguments:** fd is the file descriptor as previously defined in Section 1.3.

**Use:** The numbered lines from the specified file are inserted in line number sequence in the current program. The lines are validated on input. New lines are inserted in line number sequence. If a new line has the same line number as an existing line then the old line is replaced by the new. All variables are initialized.

Lines are read until the end of file is encountered.

**Note:** The program being merged must have been saved using the LIST command.

**Example:** Existing program XRAY

```
LIST XRAY¶
 5   Y=1
 10  PRINT
 20  FOR L=1 TO 10
 30    PRINT L;TAB(Y);"I";
 40    READ Y
 50  FOR I=1 TO Y
 60    PRINT " * ";
 70  NEXT I
 80  PRINT
 90  PRINT TAB(Y);"I"
100 NEXT L
BASIC
```

The following program file is stored on an external disk under the name TABLE:

```
200 DATA 5,4,2,3,1
300 DATA 10,15,28,15,6
999 END
```

The commands: LOAD XRAY
MERGE TABLE
add lines 200 to 999 into the existing program.

```
LIST XRAY¶
5  Y=1
10  PRINT
20  FOR L=1 TO 10
30    PRINT L;TAB(Y);"I";
40    READ Y
50    FOR I=1 TO Y
60      PRINT " * ";
70    NEXT I
80    PRINT
90    PRINT TAB(Y);"I"
100 NEXT L
200 DATA 5,4,2,3,1
300 DATA 10,15,28,15,6
999 END
```

## 6.10 NEW COMMAND

**Function:** Clears the user's program area from working storage.

**Mode:** Direct

**Format:** NEW

**Use:** Clears working storage and all variables and resets the pointers. The effect of this command is to erase all traces of the program from memory and to start over.

All open files are closed.

Use this command before typing in a new program.

**Note:** The SCR command can also be used. It works just like NEW.

**Example:**

```
|           Existing program
|
RUN
BASIC
NEW
|           Type in a new program
|
RUN         Run the second program
```

## 6.11 RENUMBER COMMAND

**Function:** Changes the line numbering in the current program.

**Mode:** Direct

**Format:**

1. REN[umber]
2. REN[umber] <1st line no.><,increment>
3. REN[umber] <1st line no.><,increment><,start line -last line>

**Arguments:** 1st line no. is required in formats 2 and 3 above and is the number to be given to the first line. The default is 10.

Increment is required in formats 2 and 3. It is the increment desired between lines. The default is 10.

Start line-last line is the range of lines to be renumbered. As format 3 shows both "1st line no." and the "increment" must be specified.

**Use:** All line references in the program will be changed according to the REN command.

Any references to line numbers in GOSUB, GOTO, IF, ON and RESUME statements are changed to the new numbers if necessary.

If any statement in the program references a line number and that line number does not exist, an error message is printed on the terminal. Renumbering is not done.

Example:

Existing Program

```
LIST¶
2 A = 1
3 B = A+2
7 PRINT A,B
10 END
BASIC
REN¶
BASIC
LIST¶
10 A = 1
20 B = A+2
30 PRINT A,B
40 END
BASIC
REN 10,5
BASIC
LIST¶
10 A = 1
15 B = A+2
20 PRINT A,B
25 END
BASIC
REN 100,20,15-25
BASIC
LIST¶
10  A = 1
100 B = A+2
120 PRINT A,B
140 END
BASIC
```

## 6.12 RUN COMMAND

**Function:** Loads and executes a Monroe BASIC program or executes the current program.

**Mode:** Direct

**Format:**
1. RUN [fd]
2. [fd]<RUN key>

**Arguments:** fd is the file descriptor as previously defined in Section 1.3.

The type specification in fd is the kind of file -
B (default) - BAC
AB - ASCBAS

Note that when the file type is omitted, the computer will look for type BAC (compressed form) first and then ASCBAS (uncompressed form).

If type "AB" (ASCBAS) is specified only ASCBAS will be searched for in the directory.

**Use:**

1. RUN
   All variables and arrays in the program area are erased and all buffers are cleared. The actions of a RESTORE statement are performed and then execution of the current program is started at the lowest numbered line.

2. RUN <vol:><filename>
   The action of a LOAD command is performed. Execution of the loaded program is then started at the lowest numbered line.

Example:

Ex. 1

```
10 READ A,B¶
20 LET A = A + B¶
30 PRINT A¶
40 DATA 2,3¶
50 END¶
RUN¶
 5
BASIC
```

If the same program is a file on the system diskette with the name APLUSB then:

Ex. 2

```
RUN APLUSB¶
 5
BASIC
```

## 6.13 SAVE COMMAND

**Function:** Creates a disk file and stores the current program into that file.

**Mode:** Direct

**Format:** SAVE <fd>

**Arguments:** fd is the file descriptor as previously defined in Section 1.3.

**Use:** The command causes the program, which is currently in the working storage, to be saved in compressed form under the given file name (type BAC). No other type can be specified. The program is saved in a compressed way to enable faster loading.

**Note:** If the file already exists on the disk the old contents in the file will be destroyed and replaced by the new program.

If the file is saved via SAVE, the file cannot be edited or listed by a utility outside of Monroe BASIC. If this is desired, refer to the LIST command.

**Example:**

```
10 - - -
|
|
4
|
999 END¶
SAVE ACT¶
BASIC
```

## 6.14 SCR COMMAND

**Function:** Clears the user's program area.

**Mode:** Direct

**Format:** SCR

**Use:** Clears working storage and all variables and also resets the pointers. The command erases all traces of the existing program from memory and starts over again.

All open files are closed. Use this or the NEW command before entering a new program.

**Example:**

```
100  ; "THIS IS A TEXT"¶
200  A = 4¶
300  ; A¶
RUN¶
THIS IS A TEST
 4
BASIC
SCR
RUN
BASIC
```

## 6.15 UNSAVE COMMAND

**Function:** Erases a file from a specified disk.

**Type:** Direct

**Format:** UNSAVE <fd>

**Arguments:** fd is the file descriptor as previously defined in Section 1.3.

Note that when the file type is omitted, the computer will look for /BAC (compressed form) first and then ASCBAS (uncompressed).

If type "AB" (ASCBAS) is specified, only ASCBAS will be searched for in the directory.

**Examples:**

Ex. 1

After the user has completed all work with file XYZ on the system disk, the file can be removed from storage by executing the following statement:

```
UNSAVE XYZ
```

Ex. 2

Erase file PROGA with file type ASCBAS.

```
UNSAVE PROGA/AB.
```

# SECTION 7
# DATA STATEMENTS

# SECTION 7
# DATA STATEMENTS

## 7.1 INTRODUCTION

Data statements consist of the set of statements shown in Table 7-1. Each data statement is described in detail following this table.

Table 7-1. Data Statements

| Statement | Description |
|---|---|
| DATA | Assigns values to variable (via READ). |
| DIM | Defines size of vector/matrix and strings. |
| DOUBLE | Designates all subsequent floating point variables and expressions to be double precision. |
| EXTEND | Specifies that spaces are significant, which allows for variable names of up to 32 characters in length. |
| FLOAT | Sets listing and input format to float mode. |
| INTEGER | Sets listing and input format to integer mode. |
| LET | Assigns a value to a variable. |
| NO EXTEND | Specifies that spaces are not significant and allows for variable names of one letter and an optional digit. |
| OPTION BASE | Defines the default minimum subscript value. |
| RANDOMIZE | Selects a random starting point for the RND function. |

| Statement | Description |
|---|---|
| READ | Assigns value(s) to variable(s). |
| RESTORE | Moves data pointer. |
| SETTIME | Sets the date and time. |
| SINGLE | Designates all subsequent floating point variables and expressions to be single precision. |

## 7.2 DATA STATEMENT

Function: Assigns values to variables; used in conjunction with READ statement.

Mode: Program

Format: DATA <value list>

Use: All DATA statements, no matter where they occur in a program, cause data to be combined into one data list. Commas are used as data separators. Single or double quotes are used to enclose items that contain a comma. If an item does not contain a comma, the enclosing quotes are optional.

A DATA statement must be the only statement on a line.

READ and DATA statements are not used without the other. See the READ statement for more information.

Examples: Ex. 1

```
10  FOR I=1 TO 3¶
20  READ A$
30  PRINT PRINT A$¶
40  NEXT I¶
50  END¶
60  DATA "HELLO: HOW ARE YOU?", "TODAY IS DEC. 13, 1980"¶
70  DATA GOOD BYE¶

RUN¶
HELLO: HOW ARE YOU?
TODAY IS DEC. 13, 1980
GOOD BYE
BASIC
```

Ex. 2

```
10  OPEN "PR:" AS FILE 1%¶
20  READ A$¶
30  PRINT #1 A$¶
40  READ A$¶
50  PRINT #1 A$¶
60  FOR I=1 TO 6¶
70    READ A$¶
80    PRINT #1 A$¶
90  NEXT I¶
100 READ A$¶
110 PRINT #1 A$¶
120 DATA ABC,DEF,GHI,JKL,MNO,PQR,STU,WXYZ¶
130 DATA ABCDEFGHIJKLMNOPQRSTUVWXYZ¶
140 END¶
RUN¶
ABC
DEF
GHI
JKL
MNO
PQR
STU
WXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
BASIC
```

## 7.3 DIM STATEMENT

Function: Defines the maximum number of elements in a vector or in a matrix. Also defines a string's maximum length.

Mode: Direct/Program

Format: 1. DIM <numeric array> [([expr1:]expr2[,...])]

2. DIM <string variable>[(expr1:]expr2[,...])][=expr3]

Arguments: Expr2 is a numeric expression specifying the maximum subscript values. The default lower limit is either 0 or 1 depending on the most recent OPTION BASE statement. The default value is 0.

An array can have any number of subscripts depending upon the available memory.

Expr3 is the maximum string length for a variable or for each of the strings in the array.

Expr1 specifies a non-default lower limit value for each subscript. It can be overridden individually for each index. This is done by replacing the single maximum index for each dimension by two values separated by a colon.

Note: A dimensioned variable can be redimensioned only if the new DIM statement defines a smaller dimension.

Use: All values used in DIM statements will be truncated to integer.

If a subscripted variable is used without appearing before in a DIM statement, it is assumed to be dimensioned to length 11 in each dimension (0-10).

The first element of every matrix is assumed to have a subscript of zero unless it is overridden by using expl or OPTION BASE. All variables have a value of zero until they are assigned a value.

Vector and matrix elements can be treated as ordinary variables in the program.

A non-dimensioned string variable's maximum length is automatically set to the current length the first time the string is assigned a non-null value. If less than 80 characters are used then a standard length of 80 characters is assigned.

Example:

```
C (1,1) = A (10,20) + B(4,7)
```

adds the two elements A (10,20) and B (4,7) into a new element C (1,1) in the matrix C.

The following alternative DIM statements for strings are available:

DIM A$(N) Defines a string vector with N + 1 strings A$(0) - A$(N). Each string has its own automatic maximum length. (See above.)

DIM A$(N)=I As above but each string's maximum length is forced to I characters.

| | |
|---|---|
| DIM A$(N,M) | Defines a string matrix with (N+1)*(M+1) strings each with its automatic maximum length. |
| DIM A$(N,M)=5 | As the matrix above but each string's maximum length is forced to 5 characters. |
| DIM A$=I | Forces the maximum length of the single string A$ to I characters. |
| DIM B$(N:M)=200 | Defines "M-(M+1)" string each with 200 bytes maximum, where M or N can be positive or negative. |
| DIM A(N:M) | Defines a vector with elements A(N) to A(M) which are totally independent of the current lower limit. |

Examples:

```
10   DIM  X(5), Z(4,3), A (10,10)
12   DIM  A4 (100)
13   DIM  A(5:10,8:20)
14   DIM  A$(20), B$(10,20)
16   DIM  C$(40) = 4
18   DIM  D$(10,10) = 8
20   DIM  Q$ = 253%
30   DIM A(-2:2) ! YIELDS VECTOR WITH FIVE
40   ! ELEMENTS A(-2),A(-1),A(0),A(1) and A(2)
50   DIM B$(-3:4)=300
```

## 7.4 DOUBLE STATEMENT

Function: Sets double precision mode. Changes all variables and expression with floating point numbers to double precision (16 digits).

Mode: Direct/Program

Format: DOUBLE

Use: The DOUBLE statement should be placed before the variables are used in the program and cannot be changed after the program has been started by RUN. This change can be made when a program line has been edited or the CLEAR or NEW command has been used. The default precision is SINGLE.

Note: Default is SINGLE. SINGLE and DOUBLE cannot be mixed in the same program.

Example:

```
NEW ¶
BASIC
10 DOUBLE¶
20 INPUT A¶
30 PRINT A¶
40 END ¶
RUN¶
? 123456789¶
 123456789
BASIC
```

## 7.5 EXTEND STATEMENT

**Function:** Specifies that spaces are significant and allows for extended length variable names.

**Mode:** Direct/Program

**Format:** EXTEND

**Use:** In the EXTEND mode, Monroe BASIC requires spaces to delimit names and functions, unless the adjoining character is a line number or an arithmetic operator (- + * /). If key words are written without spaces they may be mistaken for long variable names. Variable names can be any length up to 32 characters; all characters are significant. This will allow for more readable and understandable programs.

**Note:** The default is NO EXTEND.

**Examples:**

Ex. 1

```
AUTO¶
10 INPUT N$¶
20 EXTEND¶
30 INPUT ADDRESS$¶
40 NO EXTEND¶
50 ;N$¶
60 EXTEND¶
70 ;ADDRESS$¶
80 END¶
RUN¶
? JOHN SMITH¶
? USA¶
JOHN SMITH
USA
BASIC
```

Ex. 2

```
10 EXTEND¶
20 LET SUBTOTAL=UNITS*UNITPRICE¶
```

## 7.6 FLOAT STATEMENT

Function: Interprets all numbers without a suffix as floating point. Integers must have a "%" suffix.

Mode: Direct/Program

Format: FLOAT

Use: FLOAT is the default mode. In the FLOAT mode all variables written without the "%" suffix will be interpreted as floating point. Suppose a program was entered and listed in the INTEGER mode. If FLOAT was entered prior to the loading of this program all variables (e.g., A=12.456) would be treated as floating point variables (A%=12.456). The results of the program may change. Refer to example below and INTEGER statement for additional information.

Examples: Ex.1

```
9   OPEN "PR:" AS FILE 1%¶
10  A=12.345¶
20  B=123%¶
30  C%=B¶
40  D1%=A¶
50  PRINT #1% A,B,C%,D1%¶
60  END¶
SAVE TEXT¶
RUN¶

12.345          123             123          12
BASIC
```

Ex. 2

```
INTEGER¶
BASIC
LIST PROG ! LIST TO DISK¶
BASIC
NEW¶
BASIC
FLOAT¶
LOAD PROGB¶
LIST¶
10 REM RUN FLOAT PROGRAM AS INTEGER
20 A=12.345
30 B=123
40 C=B
50 D1=A
60 ;A,B,C,D1
70 END
RUN¶

 12            123           123           12
```

## 7.7 INTEGER STATEMENT

Function: Controls the sign suffix for integer and float variables when entering and listing programs. Allows conversion of program from float to integer.

Mode: Direct/Program

Format: INTEGER

Use: When a program is being entered and the INTEGER statement has been given, the programmer need not type the integer suffix %. On the other hand, all floating point variables should be marked by a decimal point suffix (.). The strings should have the usual $ suffix.

A program which is stored in text format and contains floating point variables can be run as an INTEGER program if the command INTEGER is given prior to loading the program. To do this, save the program via LIST, change the mode to INTEGER mode, and LOAD and RUN the program. See Example 2, below.

Note: Default is floating point format.

Examples: Ex. 1

```
100 OPEN "PR:" AS FILE 1¶
110 A.=10.532¶
120 B.=145¶
130 C=B.¶
140 D1=A.¶
150 PRINT #1 A.,B.,C,D1¶
160 END¶
RUN¶
10.532   145   145   11
BASIC
```

Example: Ex. 2

```
FLOAT¶
BASIC
LIST TEST !LIST TO DISK¶
BASIC
NEW¶
LOAD TEST¶
LIST TEST¶
100 OPEN "PR:" AS FILE 1
110  A=12.345
120  B=123
130  C=B
140  D1=A
150  PRINT #1 A,B,C,D1
160  END
RUN¶
12.345          123             123          12
BASIC
```

## 7.8 LET STATEMENT

Function: Assigns a value to a variable.

Mode: Direct/Program

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

Arguments: The use of the word LET is optional. The statement does not indicate algebraic equality but performs the calculations within the expression.

Use: The LET statement can be used anywhere in a multiple statement line.

Note: The LET keyword must be specified when assigning an extended variable name which is the same as a Monroe BASIC command name.

Example:
```
10 LET DATA=1¶
20 PRINT DATA¶
RUN¶
 1
BASIC
```

Examples:

Ex. 1
```
10  LET  A = 5.02
20  LET  X = Y7 : Z = 0
30  LET  B9 = 5 * (X/2)
40  LET  D = (3 * A) /2 * 8
```

Ex. 2
```
10  X = 36 : A = 3 + B/C : Y = X * Z
20  A$="SMITH"
30  B$="456.72"
```

## 7.9 NO EXTEND STATEMENT

Function: Disables EXTEND mode.

Mode: Direct/Program

Format: NO EXTEND

Use: In NO EXTEND mode spaces are usually ignored and variable names can only be composed of one letter and one optional digit. The default mode is NO EXTEND.

Example:

```
10  EXTEND
.
.
.
200 INPUT "NEXT NAME:"NAME$
210 INPUT "YOUR ADDRESS:"ADDRESS$
220 IF NAME$=DEFAULTNAME$ THEN 100
.
.
.
300 IF ADDRESS$=LOCATIONA$ THEN PRINT "MATCH FOUND";
301 ;NAME$,ADDRESS$
.
.
.
400 NO EXTEND
410 LET B = 400
420 INPUT "NAME IN DEFAULT" A$
.
.
.
```

## 7.10 OPTION BASE STATEMENT

**Function:** Defines the default minimum subscript value.

**Mode:** Direct/Program

**Format:** OPTION BASE <n>

**Arguments:** n must be zero or one. The default value is 0.

**Use:** Option base allows the user to specify the starting subscript for an array or vector. It allows for a saving in the memory space used for working storage when the zero element of an array is not used.

**Example:**

```
LIST¶
10  OPTION BASE 1
20  DIM A$(4)
30  A$(1)="JONES"
40  A$(2)="SMITH"
50  A$(3)="WILE"
60  A$(4)="MOHAN"
.
.
.
100 OPTION BASE 0
110 DIM B(5)
120 B(0)=1:B(1)=2:B(2)=4
130 B(3)=8:B(4)=16:B(5)=32
.
.
.
```

## 7.11 RANDOMIZE STATEMENT

**Function:** Selects a random starting value for the function RND.

**Mode:** Direct/Program

**Format:** RANDOMIZE

**Use:** This statement is placed before the first random number generator call (RND) in a program. When executed, the RND function will then select a random starting value so that if the same program is run twice, different results will be given.

**Note:** Randomize should only be used once in a program.

**Examples:**

Ex. 1

```
LIST¶
10  REM A TEST OF FUNCTIONALITY
15  REM WITHOUT RANDOMIZE STATEMENT
20  REM USING RND------FUNCTION
30  REM
35  OPEN "PR:" AS FILE 1%
40  INPUT 'HOW MANY NUMBERS?'X%
50  FOR I%=1% TO X%
60    PRINT #1% 5*RND+5
70    NEXT I%
80  END
BASIC
RUN¶
HOW MANY NUMBERS? 4
5.39556
5.22086
8.82632
9.88786
BASIC
RUN¶
```

```
HOW MANY NUMBERS? 4
5.39556
5.22086
8.82632
9.88786
BASIC
```

Ex. 2

```
10  REM A TEST OF FUNCTIONALITY OF RANDOMIZE STATEMENT¶
20  REM USING RND------FUNCTION.¶
30  REM¶
35  OPEN "PR:" AS FILE 1%¶
40  INPUT 'HOW MANY NUMBERS?'X%¶
50  FOR I%=1% TO X%¶
55    RANDOMIZE¶
60  PRINT #1%    5*RND+5¶
65  NEXT I%¶
70  END¶
RUN¶

HOW MANY NUMBERS? 4
5.05425
6.39596
8.10356
6.15174
BASIC
RUN¶
6.15174
5.85948
8.05445
5.07878
BASIC
```

## 7.12 READ STATEMENT

Function: Assigns values to variables; used in conjunction with the DATA statement.

Mode: Program

Format: READ <variable list>

Use: READ causes the variables listed to be assigned sequential values from the DATA statements. Before the program is run, Monroe BASIC creates a data block from all the DATA statements in the order they appear. Each time a READ statement is encountered in the program, the data block supplies the next value.

READ and DATA statements are used together.

If it is necessary to use the same data several times in a program, the RESTORE statement will reset the data pointer within the data block. See RESTORE statement, Section 7.

The READ and DATA statements can also be used to input string variables to a program. See Ex. 1. below.

Examples: Ex. 1

```
10  READ A$,B$,C$¶
20  PRINT A$,B$,C$¶
30  DATA CHARLIE,BOB, """STONE"""¶
RUN¶
CHARLIE          BOB             "STONE"
BASIC
```

Ex. 2

```
 50 FLOAT¶
100 READ A,B,C,D,X1,X2¶
150 DATA 3,6,1.8¶
200 DATA 6.83E-3,-86.4,3.14¶
210 PRINT "A=" A,"B=" B,"C=" C¶
220 PRINT "D=" D,"X1=" X1,"X2=" X2¶
230 END¶
RUN¶
A=3          B = 6        C= 1.8
D= .00683    X1= -86.4 X2= 3.14
BASIC
```

Note: If a comma or both leading and trailing quotation marks or apostrophes are to be read into a string, the string must be enclosed by quotation marks. This also applies to leading or trailing blanks.

## 7.13 RESTORE STATEMENT

Function: Resets data pointer to enable a specific DATA statement to be used again.

Mode: Program

Format: RESTORE [line number]

Examples:

Ex. 1

```
60  RESTORE
```

Sets the DATA statement pointer to the first DATA statement in a program.

Ex. 2

```
50  RESTORE  100
```

Sets the DATA statement pointer to the first data on line 100.

Ex. 3

```
10  READ A$¶
20  PRINT A$¶
30  READ A$¶
40  PRINT A$¶
50  FOR I=1 TO 6¶
60    READ A$¶
70    PRINT A$¶
80  NEXT I¶
90  RESTORE 120¶
100 READ A$¶
110 PRINT A$¶
120 DATA ABC,DEF,GHI,JKL,MNO,PQR,STU,WXYZ¶
130 DATA ABCDEFGHIJKLMNOPQRSTUVWXYZ¶
140 END¶
RUN¶
ABC
DEF
GHI
JKL
MNO
PQR
STU
WXYZ
ABC
BASIC
```

## 7.14 SET TIME STATEMENT

Function: Sets the date and time to the specified values.

Mode: Direct/Program.

Format: 1. SET TIME <string expression>

Arguments: String expression is of the form:
yyyy-mm-dd hh:mm:ss
where: yyyy = year, mm = month, dd = day
hh = hour (0-23), mm = minutes,
ss = seconds

Use: SET TIME is used to set the internal system clock. The TIME$ function is used to return the updated date and time. The date and time may be separated by a blank or comma. The hour, minutes and seconds may be separated by a period or colon; the TIME$ function returns a period as the separator.

Example:

```
10   PRINT "SET INTERNAL CLOCK"¶
20   PRINT "IN YYYY-MM-DD HH:MM:SS FORMAT¶
30   INPUT A$¶
40   SET TIME A$¶
:
390  ; "END TIME AND DATE"
400  ; TIME$ !DISPLAY END TIME AND DATE¶
410  END¶
RUN¶
SET INTERNAL CLOCK
IN YYYY-MM-DD HH:MM:DD FORMAT
? 1981-05-17 17:40:20¶
:
END TIME AND DATE
1981-05-17  18.05.01
BASIC
```

## 7.15 SINGLE STATEMENT

Function: Changes all variables and expressions, which are floating point numbers to single precision (6 digits).

Mode: Direct/Program

Format: SINGLE

Use: The SINGLE statement must be placed before any variables that are used and cannot be changed once the program has been started by RUN. If a line is edited or the command CLEAR is given, SINGLE may be changed to DOUBLE or vice versa. The default is SINGLE.

Note: Default is single precision. SINGLE and DOUBLE cannot be mixed in the same program.

Examples: Ex. 1

```
New¶
10  INPUT A,B¶
20  PRINT A,B¶
30  END¶

RUN¶
?12345,1234567891¶

 12345          12345678
BASIC
```

Ex. 2

```
10  DOUBLE
20  INPUT A,B¶
30  PRINT A,B¶
40  END¶
RUN¶
?12345,1234567890¶
 12345          123456789
BASIC
```

# SECTION 8
# INPUT/OUTPUT STATEMENTS

# SECTION 8
# INPUT/OUTPUT STATEMENTS

## 8.1 INTRODUCTION

Input/Output statements are program instructions that enable the user to create new disk files and perform writing, reading and maintenance operations with them. Table 8-1 lists the Input/Output statements discussed in this section.

Table 8-1. Input/Output Statements

| Statement | Function |
|---|---|
| CLOSE | Terminates I/O between the Monroe BASIC program and a peripheral device. |
| DIGITS | Sets the number of digits to be printed. |
| GET | Reads a specified number of characters from a binary file or from the console into a string variable. |
| INPUT | Fetches data from a source that is external to a program. |
| INPUT LINE | Accepts a line of input. |
| KILL | Erases a file. |
| NAME | Renames a file. |
| OPEN | Opens a file. |
| OPTION EUROPE | Allows periods and commas in "PRINT USING" output to be replaced by commas and periods, respectively, or by blanks. |
| POSIT | Positions or reads the file pointer. |
| PREPARE | Allocates and opens a new file. |
| PRINT | Writes or lists data to a specified device. |
| PRINT USING | Allows for formatted printing. |
| PUT | Writes a string to a file or the console in binary format. |

## 8.2 CLOSE STATEMENT

**Function:** Terminates input/output between the Monroe BASIC program and peripheral device(s) and closes the file(s).

**Mode:** Direct/Program

**Format:** CLOSE [channel no.,...]

**Argument:** Channel no. has the same value as in the OPEN statement and indicates the internal channel number of the file to be closed.

The CLOSE statement is used to close one or more files. If no file number is given, all files will be closed.

**Note:** The END statement closes all open files. Ordinary output with the PRINT instruction will cause the last buffer to be output when the file is closed.

All I/O operations to record I/O files are explicitly performed with the GET and PUT statements. For this reason, be sure that the user program writes explicitly the last record onto a Record I/O file before executing a CLOSE.

**Example:**

```
 5  EXTEND
10  REM CREATE AN ASCII SEQUENTIAL VARIABLE LENGTH
15  REM RECORD FILE
20  PREPARE "MAST1" AS FILE 2
30  FOR I=1 TO 5
40  READ M$,MM,DD,YY
50  PRINT #2,M$,MM,DD,YY
60  NEXT I
70  CLOSE 2
80  DATA . . .
90  DATA . . . .
..
.
.
```

## 8.3 DIGITS STATEMENT

Function: Sets the number of digits to be printed.

Mode: Direct/Program

Format: DIGITS <value>

Argument: Value is a number representing the printing accuracy.

Use: A number displayed by PRINT is rounded off to the nearest value for the last digit. Values too great to be displayed in this form are printed in exponent form with the specified number of digits.

Note: DIGITS does not affect the accuracy of calculations.

Example:

```
AUTO¶
10 INPUT A¶
20 ;A¶
30 DIGITS 2¶
40 ;A¶
50 END¶
60 ¶
BASIC
RUN¶
? 1268925¶
 1.26893E+06
 1.3E+06
BASIC
```

## 8.4 GET STATEMENT

Function: Reads one or more characters from the specified binary file or from the keyboard into a string variable.

Mode: Direct/Program

Format:
1. GET <stringvar> [COUNT bytes]
2. GET #<channel no.>,<stringvar> [COUNT bytes]

Arguments: Channel no. refers to the channel number as assigned by the OPEN or PREPARE statement.

Stringvar is the destination variable for the input transfer.

Bytes are the number of characters to be read from the console (format 1) or from a file (format 2) starting from the position of the file pointer. The default is one byte.

Use: GET is used to read a specified number of bytes from either the console or a disk file. The data is placed into a string variable and can then be processed. It is important to position the file pointer to the correction position before each GET. This is done via the POSIT statement.

Note that if GET reads from the console, user input will not be echoed on the screen. Once the correct number of characters is entered, they are processed without the return key being depressed.

Note: The fastest way to read a disk file is in blocks of 256 bytes (i.e., 256, 512, 768, etc.)

Examples:

Ex. 1

```
10  REM **USE OF GET WITHOUT COUNT**¶
20  GET B$ !GET ONE BYTE FROM KEYBOARD¶
RUN¶
X (Not shown on console)
;B$¶
X
BASIC
```

Ex. 2

```
10  GET A$ COUNT 6¶
RUN¶
AAAAAA (not shown on console)
;A$
AAAAAA
BASIC
```

Ex. 3

```
LIST¶
10  OPEN "DATA/B" AS FILE 1
20  !POSITION FILE POINTER TO 10TH BYTE
30  POSIT #1,9
40  GET #1,A$ COUNT 10
50  !PRINT 10TH TO 19TH BYTES IN FILE
60  PRINT A$
70  CLOSE
BASIC
```

## 8.5 INPUT STATEMENT

Function: Requests data from a source that is external to the program.

Mode: Direct/Program

Format:
1. INPUT #<channel no.><,list>
2. INPUT <"prompt text"> <list>

Arguments: Channel no. refers to the channel number as defined by the OPEN or PREPARE statement.

If the "#<channel no.>" is not included (see format 2 above), the system assumes data will come from the user's terminal. When the # channel number (not 0) is defined and points to another device, then the prompting function is excluded. The data is read from a file or device assigned to that specified channel. (see OPEN statement, Section 8.9). Data requested from a file must have been placed into the file by a prior PRINT statement (see Example 2.).

List contains the names of numeric variables, numeric array elements, string variables or string array elements. As with the PRINT statement, if a semicolon follows the last list item, the carriage return and line feed normally supplied when a terminator is typed is suppressed. If a comma follows the last list item, the cursor is moved to the next 14-column tab position when a terminator is typed.

Prompt text is a character string delimited by quotes. When included prompting messages can be specified to query the user for required information.

Use: During program execution, the programmer can enter data when prompted. INPUT (format 2 above) causes the terminal to pause during execution, print the prompt text and wait for the user to enter data. If no prompt text is included a question mark is displayed on the screen.

The user then enters the values separated by commas. The values are stored. If insufficient data is given or too much data is entered, the system displays error message No. 148 or 150, respectively (see Appendix B) and no variables are updated. Depending upon how many values are to be accepted by the INPUT command, the programmer may include a PRINT statement that reminds the user of the kind of input required. This is conveniently done with the multiple format shown in example 3, below.

Examples:

Ex. 1

```
AUTO¶
10  INPUT A,B,C¶
20  ;C,A,B¶
30¶
BASIC
RUN¶
? 1,2,3¶
3              1              2
BASIC
```

Ex. 2

```
LIST¶
10  PREPARE "FILEA" AS FILE 1
20  INPUT A,B,C
30  ;#1,A","B","C¶
40  CLOSE #1! WRITES END OF FILE
50  OPEN "FILEA" AS FILE 1
60  INPUT #1, X,Y,Z
70  ;X,Y,Z
RUN¶
? 5,7,9
5              7              9
BASIC
```

Ex. 3

```
10  INPUT "YOUR NAME : ?"A$
20  INPUT "YOUR ADDRESS : ?"B$
```

Is equivalent to

```
10  PRINT "YOUR NAME : ";
20  INPUT A$
30  PRINT "YOUR ADDRESS : ";
40  INPUT B$
```

Ex. 4

```
20  OPEN "MAST" AS FILE 3
30  INPUT #3, A$
.
.
.
```

## 8.6 INPUT LINE STATEMENT

Function: Accepts a line of input from the terminal or file.

Mode: DIRECT/PROGRAM

Format:
1. INPUT LINE <string variable>
2. INPUT LINE [#channel no.],<string variable>

Arguments: Channel no. is associated with the OPEN statement and stands for a device or file as a logical unit.

String variable is any legal string variable where the text from the keyboard or from a specified file is placed.

Use: INPUT LINE causes the program to accept a line of characters from the terminal or from the specified ASCII file.

All characters belonging to the line are read - spaces, punctuation characters, and quotes. The line termination characters, carriage return (CR) and line feed (LF) are always appended to the string regardless of the actual terminator character; the terminator character is not echoed. (See Section 2.13.)

No text string (prompting message) can be written with the INPUT LINE statement; this facility is only available in the INPUT statement. The PRINT statement can be used to print out the prompt text.

Examples:

Ex. 1

```
10  ;"YOUR ADDRESS? ";¶
20  INPUT LINE A$¶
30  PRINT TAB(25);A$¶
RUN¶
  YOUR ADDRESS? MORRIS PLAINS, NJ¶
                              MORRIS PLAINS,NJ
BASIC
```

Ex.2

```
LIST¶
10  INPUT LINE A$
20  B$=LEFT$(A$,LEN(A$)-2)
BASIC
```

Line 20 removes CR and LF from string A$.

## 8.7 KILL STATEMENT

Function: Erases the file, named by the string, from the user's file area.

Mode: Program/Direct

Format: KILL <string expression>

Arguments: <string expression> contains the file descriptor (as previously defined in Section 1.3) for the file to be erased. The default file type is /A-ASCII.

Note: A user is not allowed to KILL a file if the volume is write-protected.

Example:

```
LIST¶
10  !
20  ! THIS IS A SIMPLE EXAMPLE OF A
30  ! BACKUP PROCEDURE USING NAME
40  ! AND KILL STATEMENTS.
50  !
55  !
57  INPUT "FILE TO BE BACKED UP? "C$
60  OPEN C$ AS FILE 1
70  PREPARE "DUMMY" AS FILE 2
80  PRINT #2, TIME$ ! PRINT TIME FIRST ON FILE.
90  ON ERROR GOTO 140
100 INPUT #1,A$ ! READ FROM C$
110 PRINT #2,A$ ! WRITE TO DUMMY
120 GOTO 100
130 !
140 IF ERRCODE<>14 THEN STOP ! STOP IF NOT EOF.
150 CLOSE
160 KILL C$ ! DELETE OLD FILE.
170 !
180 ! RENAME NEW FILE TO THE OLD NAME.
190 !
```

```
200  NAME "DUMMY" AS C$
210  CLOSE
220  END
BASIC
```

## 8.8 NAME STATEMENT

Function: Renames a file on disk.

Mode: Program/Direct.

Format: NAME <string expression1> AS <string expression2>

Arguments: <string expression1> specifies the name of the file you want to rename.

<string expression2> is the new name.

Note that type in the file descriptor is the type of any file. If no type is specified A-ASCII is the default. If type2 is specified, it must be different than type1. If type1 is specified and not type2, then the file type will not be changed.

Refer to the 8800 Series Monroe Operating System Programmer's Reference Manual for available types.

Note: If the file being renamed is referenced by another program when the NAME statement is executed, an error messsage will result.

Examples: Ex. 1

```
100 NAME "DRO:OLD" AS "NET"
```

Ex. 2

The following statement:

```
200 NAME "DRO:ABC/B" AS "XYZ"
```

Changes the name of the file ABC/B on disk DRO:. The NAME statement cannot transfer a file from one device to another.

Ex. 3

```
120 NAME "NJTT" AS "NJTT1"
```

changes name of file NJTT to NJTT1 on the system volume.

## 8.9 OPEN STATEMENT

Function: Opens a device or a file for sequential access with a channel number internal to a Monroe BASIC program.

Mode: Direct/Program

Format: OPEN <string expr> AS FILE <channel no.> [MODE a% + b%]

Note: The above format without the MODE option opens a file in MODE 192, a byte I/O, sharable read, exclusive write (SREW) mode. To use OPEN for other I/O modes and random access refer to Section 14, Advanced Programming. It is important to note that when accessing a write-protected disk, the MODE option must be included (i.e., "MODE 192%" for sharable read and "MODE 193%" for exclusive read).

Arguments: String expr is the name of the disk file to be opened. If no file type is specified the default is /A-ASCII.

Channel no. after AS FILE must be an integer value between 1 and 250 corresponding to the internal channel number on which the file is opened.

a% + b%, used with MODE option, is defined in Section 14.

Use: OPEN is used to open files which already exist. When more than a few items are to be read or written, then the technique used by the READ, DATA and INPUT statements is inefficient. When a sequence of data items is to be transferred, the data can be conveniently handled as a data file through the use of a "channel".

The data file has both an external name by which it is identified within the system and a channel number that references the file. The OPEN statement associates the external file specification with the internal channel number.

The channel number is referred to by use of the symbol # (number sign) and is followed by the channel number.

Writing and reading from a file is done by use of INPUT and PRINT statement of a special form. The PRINT and INPUT formats to be used with the OPEN statements are:

```
PRINT # <channel no.>,[<list>]
INPUT # <channel no.>, <list>
INPUT LINE # <channel no.>, <list>
```

The <channel no.> is the same value as the expression in the OPEN statement <channel no.> and the <list> is a list of variable names, expressions, or constants as described in the PRINT and INPUT statement descriptions.

GET and PUT can also be used to read from and write to a binary file respectively.

Note: A file must already be created using the PREPARE statement before OPEN can be used. When data is to be read from an existing file, the file should be opened by the OPEN instruction.

Examples:

Ex. 1

```
50  OPEN "TEST" AS FILE 1
```

Ex. 2

```
10  OPEN "DATA" AS FILE 2
20  INPUT #2,A
30  INPUT #2,B
40  INPUT #2,C7$
```

The values of the variables A,B, and C7$ are read from the file, which was opened as file number 2. The values are read directly after the values last read. If reading is to be done from the beginning of the file, it must be opened again with the OPEN instruction.

Ex. 3

```
30  INPUT A,B,C$¶
40  ;A,B,C$¶
50  PREPARE "DATA" AS FILE 2¶
60  PRINT #2, C$;",";A;",";B
65  ! STORED IN FILE AS -HELLO,12,24¶
70  CLOSE 2¶
80  A=0:B=0:C$=" "¶
90  ;A,B,C$¶
100 OPEN "DATA" AS FILE 4¶
110 INPUT #4, D$,E,F¶
120 ;E,F,D$¶
130 CLOSE 4¶
140 END¶
RUN¶
 ? 12,24,HELLO¶
 12            24          HELLO
 0             0
 12            24          HELLO
BASIC
```

## 8.10 OPTION EUROPE STATEMENT

Function: Replaces periods and commas in "PRINT USING" output by commas and periods respectively.

Mode: Direct/Program.

Format: OPTION EUROPE n

Arguments: n can be either 1 or 0. A value of 1 replaces periods and commas as previously specified while a value of 0 (default) negates the replacement.

Use: This statement is used before the PRINT USING statement to allow output to conform to European notation. That is, commas in numbers are replaced by periods and periods by commas.

Example:

```
LIST¶
5  OPEN "PR:" AS FILE 1
10  DOUBLE
20  A=1.23456789E+06
30  ; #1 "FORMAT:"
40  A$="########.## ##,###,###.## #%###%###.##"
50  A$=A$+" ##########~## #,###,###~## #%###%##~##"
60  ; #1 A$
70  FOR I=0 TO 1
80    OPTION EUROPE I
90    ; #1 USING "OPTION EUROPE = #",I
100    ; #1 USING A$,A,A,A,A,A,A
110  NEXT I
120  END
RUN¶

FORMAT:
########.## #,###,###.## #%###%###.## ##########~##
  #,###,###~## #%###%###~##
OPTION EUROPE = 0
1234567.89 1,234,567.89 1 234 567.89 1234567 89
1,234,567 89 1 234 567 89
OPTION EUROPE = 1
1234567,89 1.234.567,89 1 234 567,89 1234567 89
1.234.567 89 1 234 567 89
```

## 8.11 POSIT STATEMENT

Function: Positions the file pointer or reads where file pointer is positioned.

Mode: Program/Direct.

Format:

1. Positions File Pointer
   POSIT # <channel no.>,<position>
2. Reads File Pointer
   POSIT (<channel no.>)

Arguments: Channel no. corresponds to the internal channel number on which the file is opened.

Position is the number of bytes from the beginning of the file where access is to begin. The position supplied or returned is a floating point number.

Use: Each data file contains a pointer specifying the present position in bytes from the beginning of the file. This pointer can be read or positioned to a specific byte position using POSIT.

Format 1, above, is used to move the file pointer the specified number of positions from the beginning of the file (the first position). The first position = 0. POSIT can be used together with all file handling instructions. The instruction points to a certain character or the first character of a character sequence to be read or written.

Format 2, above, yields the current position of the file pointer.

Note: To use POSIT for record I/O refer to Section 14, "Advanced Programming".

Examples:

Ex. 1

```
.
.
.
80 POSIT #1,15¶
```

The file pointer is moved to position 15 (i.e. it points to the 16th character of file number 1).

Ex. 2

```
.
.
.
50 A=POSIT(1)¶
```

A=the position of the file pointer. In Example 1 above, the file pointer is in position 15, i.e. A=15.

Ex. 3

```
(Read - byte I/O)
LIST¶
10  OPEN "VOL1:LISTER" AS FILE 2
20  POSIT #2, 10
30  GET #2, A$ COUNT 5
40  PRINT A$
BASIC
```

Ex. 4

```
(Position - Byte I/O)
LIST¶
10  PREPARE "TEST" AS FILE 3
20  ;POSIT(3)
30  ;#3"JOHN ALDER";
40  ;POSIT(3)
RUN¶
0
10
BASIC
```

## 8.12 PREPARE STATEMENT

Function: Allocates and opens a new file with an internal file number within the current program.

Mode: Direct/Program.

Format: PREPARE <string expr> AS FILE <channel no.>
[MODE a% + b%]

Note: The above format without the MODE option opens a file in MODE 192, a byte I/O, sharable read, exclusive write (SREW) mode. To use PREPARE for other I/O modes and random access refer to Section 14, Advanced Programming. It is important to note that when accessing a write-protected disk, the MODE option must be included (i.e., "MODE 192%" for sharable read and "MODE 193%" for exclusive read).

Arguments: String expression corresponds to an external file specification for the file to be created and opened. If no file type is specified, the default is /A-ASCII.

Channel no. after AS FILE must be an integer value between 1 and 250 corresponding to the internal channel number on which the file is opened.

Use: PREPARE performs the same function as OPEN with the exception that it first creates the file. If the file exists before PREPARE, it will be deleted and a new file established. The use of OPEN assumes that the file exists.

Examples:

Ex. 1

```
LIST¶
 10  REM *EXAMPLE OF PREPARE*
 20  INPUT "ENTER NEW FILENAME? " A$
 30  PREPARE A$ AS FILE 2
 40  INPUT "5-LETTER MERCHANDISE CODE? " B$
 50  INPUT "COLOR CODE (R-RED,G-GREEN,B-BLUE)? "C$
 60  INPUT "NUMBER REQUIRED (1K,5K, OR 9K)? "D$
 70  INPUT "SOURCE CODE (1,2 OR 3)? " E$
 80  PRINT #2 B$,C$,D$,E$
 90  PRINT B$,C$,D$,E$
100  INPUT "CONTINUE (Y OR N) ? " F$
110  IF F$ = "Y" GOTO 40
120  CLOSE #2
130  END
BASIC
```

Ex. 2

```
 10 REM--------TESTING THE USE OF PREPARE STATEMENT---¶
 20 REM THIS PROGRAM CREATES A FILE ON THE DISK-¶
 30 PREPARE "PASC:NEWFILE" AS FILE 3%¶
 40 A$="AB"¶
 50 B$="CD"¶
 60 C$="EF"¶
 70 PUT #3% A$+B$+C$
 80 POSIT #3,0¶
 90 GET #3,D$ COUNT 6¶
100 ;D$¶
RUN¶
ABCDEF
BASIC
```

## 8.13 PRINT STATEMENT

Function: Prints data on a device in ASCII format.

Mode: Direct/Program.

Format:
1. PRINT
2. PRINT <list>
3. PRINT #<channel no.> <list>

Note: The keyword PRINT can be replaced in the above formats with a semicolon (e.g., ;<list>).

Arguments: Channel no. corresponds to the channel number in the statement. If omitted, the list data will be displayed on the screen.

List can contain variables, expressions or text strings. If an element in the PRINT list is not a simple variable or a constant, the expression is evaluated before the data is printed. Text strings are enclosed in quotes.

Use: The positions on a line are numbered from 0 to 39/79 (39 is valid on education model and 79 on business model). The line is subdivided into columns, fixed tabulator positions, starting in positions 0, 15, 30, 45, 60, and 75. A comma (,) after a variable or a string in the PRINT list specifies that the next element of the list will be printed in the next column. Two commas together in a PRINT statement cause a column to be skipped.

A semicolon (;) following a variable or a string in the list causes the next element in the list to be printed in the position immediately after the previous character. If neither a comma or semicolon separates list items, a semicolon is assumed. If the list is

terminated by a semicolon (;) no carriage return or line feed will follow the last item printed.

A PRINT statement without any argument causes a carriage return and line feed to be printed (i.e. one blank line).

When a line is filled, the display continues on the first portion of the next line.

The TAB(con) and CUR(y,x) functions are used to cause data to be printed in certain positions. These functions instruct Monroe BASIC where to print the next value of the PRINT list. Note that for TAB, if the cursor is already beyond this point, it moves to the corresponding position on the next row.

If the #<channel no.> is not written, the system assumes the user's terminal. The data is output to a file or device assigned to the specified channel.

Examples:

Ex. 1

```
110  PRINT X;Y; 5
120  PRINT                    (spaces one line)
130  PRINT "VALUE= ";X3, " SAM2= ";A+2
```

Ex. 2

```
10  LET A = 5
20  LET B = 2
30  PRINT A,B,A+B,A*B,A-B,B-A,A/B
40  END
```

Ex. 3

```
110  PRINT TAB(2)  B; TAB(2*R)  C
```

Ex. 4 Cursor positioning (Console Only)

```
PRINT CUR(5,12) "TESTSTRING";
```

Writes TESTSTRING beginning at row 5 and column 12. The ";" in the PRINT statement above specifies that no carriage return or line feed will follow.

Ex. 5

```
100  OPEN "MYFILE" AS FILE 2
200  PRINT #2, A","B","C
```

Opens a disk file by the name MYFILE. Values are written (printed) to this file.

## 8.14 PRINT USING STATEMENT

Function: Specifies the appearance (format) of printed data.

Mode: Direct/Program.

Format: See Section 11.

Use: See Section 11.

Example:
```
10  PRINT USING "$##.##";12.34
RUN¶
$12.34
BASIC
```

## 8.15 PUT STATEMENT

Function: Writes a string to a file or console in binary format.

Mode: Direct/Program.

Format: PUT [#channel no.,]<string>

Arguments: Channel no. refers to the channel number previously defined by an OPEN or PREPARE statement.

String is either a string variable or a string expression.

Use: PUT is used to write in binary format a string variable or expression to a file or the console. POSIT is used to position the file pointer to the desired point in the file.

Example:

```
LIST¶
10  PREPARE "FILA/B" AS FILE 2
20  ! FILEA/B SPECIFIES BINARY DATA FILE
30  INPUT "ENTER DATA? " A$
40  PUT #2%,A$
50  POSIT #2,0
60  GET #2,B$ COUNT 10
70  ;B$
RUN¶
 ENTER DATA? "JOHN SMITH"¶
 JOHN SMITH
BASIC
```

# SECTION 9
# PROGRAM CONTROL STATEMENTS

# SECTION 9
# PROGRAM CONTROL STATEMENTS

## 9.1 INTRODUCTION

In previous sections, program examples have been executed top to bottom in order of their line numbers. In most applications, however, a programmer needs the flexibility of specifying alternate execution routes. For example, branching from one point of a program to another or reexecuting a given set of code for a specifying number of times may be required. Control statements are the mechanism which allows the programmer to control the flow of a program, and to interrupt and resume sequential execution at will. Some of these statements may also be used for debugging. Table 9-1 lists the control statements discussed in this section.

Table 9-1. Program Control Statements

| Statement | Function |
|---|---|
| BYE | Transfers control from Monroe BASIC to the Operating System. |
| CHAIN | Loads and executes a program from a program currently being executed. |
| COMMON | Transfers variables to the next CHAINed program. |
| DEF | Defines single or multi-line user defined functions. |
| END | Terminates execution of a Monroe BASIC program. |
| FNEND | Terminates a multi-statement function definition. |
| FOR | Provides the specifications for repetition in a program loop. |
| GOSUB | Directs program control to the first statement of a subroutine. |

Table 9-1. Program Control Statements (Cont.)

| Statement | Description |
|---|---|
| GOTO | Transfers control unconditionally to the statement with the specified line number. |
| IF...THEN...ELSE | Executes a specified statement or transfers control to another line depending upon a stated condition. |
| NEXT | Denotes the end of a FOR loop. |
| NO TRACE | Disables trace mode. |
| ON ERROR GOTO | Specifies a user routine for error handling. |
| ON...GOSUB.. | Transfers control conditionally to one of several subroutines or to entry points to one subroutine. |
| ON...GOTO | Transfers control to one of several lines depending on the value of the expression at the time the statement is executed. |
| ON...RESTORE | Restores the DATA pointer to one of several lines in the program. |
| PAUSE | Pauses the current BASIC program task. |
| ON...RESUME | Transfers control to one of several places in error handling situations. |
| RESUME | Transfers control from an ERROR subroutine. |
| RETURN | Transfers control in a subroutine back to the calling GOSUB or causes a return from a multi-line function. |

Table 9-1. Program Control Statements (Cont.)

| Statement | Description |
|---|---|
| STOP | Stops program execution. |
| TRACE | Prints line numbers of designated executed program line. |
| WEND | Defines the limit of the WHILE loop. |
| WHILE | Defines the specific condition for leaving a loop. |

It is important to note that transfer of program control is not allowed from inside a function definition or subroutine to another part of a program. Also, transfer is not allowed from any point in a program to inside the body of a function definition or subroutine. If such an attempt is made, a run-time error will result in BASIC versions R1-04 or later.

## 9.2 BYE STATEMENT

**Function:** Finishes the working session in Monroe BASIC and returns control to the operating system.

**Mode:** Direct/Program

**Format:** BYE

**Action:** BYE closes and saves any files remaining open for that user and returns control to the operating system.

**Examples:**

1.
```
BASIC
BYE¶
-
```

2.
```
|                          |
|        EXISTING PROGRAM  |
|                          |
|                          |
100 BYE¶
-
```

## 9.3 CHAIN STATEMENT

Function: Loads and executes a program.

Mode: Direct/Program

Format: CHAIN <string expression>

Arguments: String expression contains the file descriptor specifying the name of a disk file from which a Monroe BASIC program is to be loaded. The default file type is /B-BAC. See Section 1.3 for a definition of the file descriptor.

Use: If the user program is too large to be loaded into memory and run in one operation the user can segment the program into two or more separate programs. The CHAIN instruction is used as a logical termination of one program to call the next one. Each program is called by its name. The program in the computer is erased and the new one is loaded. The lowest numbered program line is executed first as though a RUN command had been used. The CHAIN instruction is the last instruction to be executed. The last program in a chain does not need any CHAIN statement, but control is often transferred by CHAIN back to a program that allows the user to select the program to be run.

When CHAIN is executed, all open files for the current program are closed. Any files to be used in common by several programs should be opened in each program.

Note: Variables can be passed on to a CHAINed program by means of the COMMON instruction.

Example:

```
BASIC
NEW¶
AUTO¶
10 COMMON A,B$=10,C¶
20; "Aƀ=";A¶          (Note: ƀ = space)
30; "B$=ƀ";B$¶
40; C+A¶
50 END¶
60¶
BASIC
SAVE TESTA¶
BASIC
NEW¶
BASIC
AUTO¶
10 COMMON A,B$=10,C
20 INPUT A¶
30 INPUT B$¶
40 INPUT C¶
50; A,B$,C¶
60 CHAIN "TESTA"¶
70¶
BASIC
SAVE TEST¶
BASIC¶
RUN¶
?1¶
?A¶
?2¶
  1              A              2
A=1
B$=A
3
BASIC
```

## 9.4 COMMON STATEMENT

Function: Enables variables to be passed from one program to another when the programs are CHAINed together.

Mode: Program

Format:
1. COMMON <numeric array>[(expr1:]expr2[,...])]
2. COMMON <string variable>[([expr1:]expr2[,...])] [=expr3]

Arguments: Expr2 is a numeric expression specifying the maximum subscript values. The default lower limit is either 0 or 1 depending on the most recent OPTION BASE statement. The default value is 0.

The array can be any number of subscripts depending upon the available memory.

Expr3 is the maximum string length for a variable or for each of the strings in the array.

Expr1 specifies a non default lower limit value for each subscript. It can be overridden individually for each index. This is done by replacing the single maximum index for each dimension by two values separated by a colon.

Use: The variables being passed must be present in the COMMON statement and the COMMON statement(s) must be executed before any other statement. The passed variables must look alike in all programs where the common variables are to be used.

Note: The length of common string variables must be declared. The COMMON statement replaces the DIM statement.

Common variables will not be passed from a compressed BASIC program (Type BAC) to an ASCII Monroe BASIC program (ASCBAS).

Example:
```
10 COMMON A%,D(8),F$(20)=40
```

Also, see example for CHAIN, Section 9.

## 9.5 DEF STATEMENT

**Function:** Defines single and multiple line user-defined functions.

**Mode:** Program

**Format:**
1. DEF FN<name>[type][(arguments)]=<expression>
2. DEF FN<name>[type][(arguments)][LOCAL variables]

**Arguments:** <name> is any valid variable name.

Type is optional and can be either % (or ".") or $.

Arguments consist of dummy variables. They are optional. If included, the same number of arguments must appear in the function reference.

Expression can be any valid arithmetic or string expression which matches the type of the function.

Variables are temporary variables needed within a function definition. The LOCAL keyword makes possible the local variable name option. Variables should be declared local to the function in order to protect the global variables from being disturbed. This eliminates the need for using different variable names outside the function. Arrays cannot be declared LOCAL. String variables specified as LOCAL must have an explicit length. (See Example below.)

**Note:** Type will default to either FLOAT or INTEGER if not specified; which one depends on the current mode, FLOAT or INTEGER. Make sure when editing or revising the program that the type in effect is consistent with type in your program. Otherwise, erroneous results may occur.

Use:

Monroe BASIC allows the programmer to define user functions and call these functions in the same manner as standard functions such as SIN. User-defined functions can consist of a single line (see format 1, above) or multiple lines (see format 2). A multiple line DEF function differs from the single line functions due to the absence of an equal sign following the function designation on the first line. Any number of arguments of any type or any mixture of types may be used including zero. Within the multiple line function definition there must be statements of the form: RETURN <expression> and FNEND.

When the RETURN statement is encountered, the expression is evaluated and used as the value of the function, and exit is performed from the definition. The definition may contain more than one RETURN statement, as can be seen from the example 2 below.

Multiple line DEF functions can be called recursively; one multiple line function definition can refer to itself or another multiple line function definition. The same rules apply here as for the nesting of program loops. There must be no transfer from within the definition to outside its boundaries or from outside the definition into it. The line numbers used by the definition must not be referred to elsewhere in the program. If ON ERROR GOTO is used inside the function it will be reset to the line number that was active in the calling program when the function exits.

If temporary variables are needed within a function definition they should be declared local to the function in order to protect the global variables from being disturbed. The LOCAL modifier makes possible the local variable name option. See example 4 below.

Examples:

Ex. 1

Single Line Function:

```
10  DEF FNA(X,Y)=X+X*Y¶
```

Ex. 2

Multiple Line Function: The function below determines the larger of two numbers and returns that number. Such use of the IF - THEN instruction is frequently found in multiple line functions:

```
10  DEF FNM(X,Y)¶
20    IF Y<=X THEN RETURN X¶
30    RETURN Y¶
40  FNEND¶
```

Ex. 3

Multiple Line Function: This example shows a recursive function that computes the N-factorial. (However, there are more efficient, non-recursive routines for the computation of N-factorial.):

```
LIST¶
 5  EXTEND
10  DEF FNFAK(M%)
20    IF M%=0% THEN RETURN 1% ELSE RETURN
       M%*FNFAK(M%-1%)
30  FNEND
32  REM FACTORIAL FACTOR MUST BE <9
35  INPUT "VALUE FOR FACTORIAL (<9)?: "X
40  PRINT X;"-FACTORIAL EQUALS ";FNFAK(X)
50  END
RUN¶
  VALUE FOR FACTORIAL (<9)?: 4¶
  4-FACTORIAL EQUALS 24
BASIC
```

Ex. 4

This example shows the use of the LOCAL option.

```
LIST¶
10  DEF FNA(X) LOCAL A,A$=10
20  A=33: A$="LOCAL"
30  PRINT A$
40  PRINT A
50  RETURN 5*X
60  FNEND
100  A=22: A$="GLOBAL"
110  PRINT A$
120  PRINT A
130  PRINT FNA(8)
RUN¶
GLOBAL
  22
LOCAL
  33
  40
BASIC
```

Ex. 5

The next example shows a string function:

```
LIST¶
100  PRINT FNV1$("AABBCCDDEEFF",5%,10%)
110  END
120  DEF FNV1$(A$,B%,C%)
130    IF B%=C% THEN RETURN LEFT$(A$,B%) ELSE
        RETURN RIGHT$(A$,C%-B%)
140 FNEND
RUN¶
 CCDDEEFF
BASIC
```

## 9.6 END STATEMENT

**Function:** Terminates a Monroe BASIC program.

**Mode:** Program

**Format:** END

**Use:** The END instruction is normally, but need not be, the last statement of a program. After END there must be only subroutines and functions which can be called by the main program. END closes all files.

**Note:** The variables keep their values after END. END should be on a line by itself.

**Example:**

Ex. 1

```
10  REM **¶
20  A$="5000"¶
30  OPEN "XRAY" AS FILE 1¶
40  PUT A$¶
50  CLOSE¶
60  END¶
```

Ex. 2

```
NEW¶
*BASIC*
AUTO¶
10  READ A,B,C¶
20  IF A=99 GOTO 60¶
30   ;A;B;C;¶
40   GOTO 10¶
50   DATA 4,5,6,1,2,3,99,99,99¶
60   END¶
70  ¶
BASIC
```

## 9.7 FNEND STATEMENT

Function: Terminates a multiple statement function definition.

Mode: Program

Note: This statement must never be reached by sequential statement execution. The function definition should be exited before this statement by a RETURN <expr>.

Example:

```
LIST¶
10  DEF FNMOT (X,Y)
20  IF Y >=X**3 THEN RETURN X
30  RETURN Y
40  FNEND
BASIC
```

## 9.8 FOR STATEMENT

Function: Sets up program loops by causing the execution of one or more statements for a specified number of times. NEXT statement is also necessary.

Format:
```
FOR <variable> = <expression> TO <expression>
    [STEP expr]
NEXT <variable>
```

Mode: Program

Arguments: The variable in the FOR ... TO statement is initially set to the value of the first expression. The statements following the FOR are then executed. When the corresponding NEXT statement is encountered, the variable is incremented by the value indicated as the STEP interval. The NEXT statement is specified separately. See NEXT.

Use: The initial value is assigned to the variable and the loan test is made. When the step value is positive and the variable value exceeds the TO expression, then the next statement executed is the one following NEXT.

If the step value is negative and the expression exceeds the variable, the next statement executed is the one following NEXT. Otherwise, the program will execute statements between FOR and NEXT and repeat the test.

Program execution for a step value of 0 does not end in the normal manner. It continues until interrupted by the user.

The expressions within the FOR statements are evaluated once upon initial entry to the loop. The test for completion of the loop is made prior to each execution of the loop.

Program loops have four characteristic parts:

1. Initialization to set up the conditions which must exist for the first execution of the loop.

2. The body of the loop to perform the operation to be repeated.

3. The modification which alters some value and makes each execution of the loop different.

4. The termination condition, an exit test which, when satisfied, completes the loop. Execution continues to the program statement following the loop.

If the STEP expression is omitted from the FOR statement, +1 is the assumed value. Since +1 is a common STEP value, that position of the statement is frequently omitted.

The control variable can be modified within the loop. When control falls through the loop, the control variable retains the last value used within the loop plus the step value.

FOR loops can be nested but not overlapped. Nesting is a programming technique in which one or more loops are completely within another loop. The depth of nesting depends upon the amount of user memory space available.

The field of one loop must not cross the field of another loop.

It is possible to leave a FOR NEXT loop without the control variable reaching the termination value. A conditional or unconditional transfer can be used to exit from a loop. When reentering a loop which was left earlier without being completed remember that the terminator and STEP values are not reevaluated.

Note: The FOR statement is especially suited for using integer variables; it results in faster loop execution.

Examples: Ex. 1

This program demonstrates a FOR - NEXT loop. The loop is executed 20 times. When the value for A is 20, control leaves the loop and displays the last value of A. A STEP value of +1 assumed since FOR contains no STEP variable.

```
10   FOR A%=1% TO 20%¶
20   PRINT "A="; A%¶
30   NEXT A%¶
40   PRINT "A="; A%¶
RUN¶
A=1
A=2
.
.
.
A=21
BASIC
```

The loop consists of lines 10, 20 and 30. The numbers A=1 to A=20 are printed when the loop is executed. After A=20, control passes to line 40 which causes A=21 to be displayed.

Ex. 2

Acceptable nesting

```
20 FOR  A = 1 TO 10
30   FOR  B = 2 TO 11
40   NEXT B
50   FOR  C = 1 TO 10
60   NEXT C
70 NEXT A
```

Unacceptable nesting

```
100 FOR A = 1 TO 10
110   FOR B = 2 TO 11
120   NEXT A
130 NEXT B
```

## 9.9 GOSUB STATEMENT

Function: Transfers control to the first of a sequence of statements that form a subroutine.

Mode: Program

Format: GOSUB <line no.>

Arguments: Line no. is the first line number of the called subroutine. Control is transferred to that line in the subroutine.

Use: A subprogram is a sequence of instructions which perform a task that may be repeated several times in a program. To call such a sequence of instructions, Monroe BASIC provides subroutines and functions.

A subroutine is part of a program that received control upon execution of a GOSUB statement. Upon completion of the subroutine a RETURN statement is used to exit the subroutine and continue program execution. At this point control is transferred to the statement following the GOSUB statement.

The only instruction that may be used to exit a subroutine is RETURN.

Example:

```
.
.
.
150  GOSUB 1300
.
.
.
```

```
300  GOSUB 1300
.
.
.
400  GOSUB 1800
.
.
.
1300  REM ** SUBROUTINE #1**
1310  FOR I = J TO K
1320    LET I = 2 * N
1330    PRINT I
1340  NEXT I
1350  RETURN
.
.
.
1800  REM ** SUBROUTINE # 2+XX
.
.
.
1900  RETURN
2900  END
```

## 9.10 GOTO STATEMENT

Function: Transfers program execution unconditionally to a specified program line.

Mode: Direct/Program

Format: GOTO <line no.>

Arguments: Line no. is usually not the next sequential line in the program. GOTO must be written as one word in EXTEND mode.

Use: The GOTO statement is used when it is desired to unconditionally jump to a line other than the next sequential line in the program. It is possible to jump backward as well as forward within a program.

When written as a part of a multiple statement line, GOTO should always be the last statement on the line, since any statement following the GOTO statement on the same line will never be executed.

Note: The GOTO statement can be used in the direct mode after a pause, i.e., STOP or CTRL C.

Example:

```
.
.
.
110  X = 20
120  PRINT X
130  X = X + 1
140  IF X = Z THEN 900
150  GOTO 120
.
.
.
900  END
```

## 9.11 IF...THEN...ELSE STATEMENT

Function: Executes a specified statement depending upon a stated condition.

Mode: Program

Format:

IF <condition> [THEN / GOTO] <argument1> [ELSE argument2]

Arguments: Condition is a relational expression. The test of whether or not a given condition is true is normally performed by means of relational operators. They permit comparisons to be performed that determine the relationship of variables, constants, or expressions to each other. Note that condition may not reference the same function (e.g., FNA$(x) < > FNA$(y)).

The result of the comparison is a numerical value indicating whether a given relationship between two data items is true or false.

Argument1 can be a line number or one or more statements. The line number when present is the line number control is transferred to when the condition (relational expression) is evaluated to be true (<>0). The statement when present is any Monroe BASIC statement(s) which is executed when the condition (relational expression) is evaluated to be true (<>0).

Argument2 can be a line number or one or more statements. This line number when present (requires ELSE keyword) is the line number control is transferred to when the condition (relational expression) is evaluated to be false (0).

The statement(s) when present (requires ELSE keyword) is any Monroe BASIC statement(s) which is executed when the condition (relational expression) is evaluated to be false (0).

Note: THEN may be replaced by GOTO in the format but the arguments are then restricted to line numbers only.

Use: IF...THEN...ELSE is a built-in test which allows a program to determine which of two routes it should choose during execution.

The specified condition is tested. If the condition is met (the expression is logically true), control is transferred to the line number given after THEN or the statement given after THEN is executed. If the condition is not met (the expression is logically false), the program execution continues at the program line following the IF statement if the "ELSE" clause is not included.

THEN may be followed by either a line number or one or more Monroe BASIC statements. If Monroe BASIC statements are given and the condition is met, these statements will be executed before the program continues with the line following the IF statement. The condition applies to all statements that follow on the same line as the IF statement.

ELSE, when included, is followed either by a line number which is used as a jump address or one or more statements which are executed before the line following the IF statement.

IF statements can be nested, but they must all fit on one program line.

When relational expressions are evaluated, the arithmetic operations take precedence in their usual order. The relational operators have equal weight and are evaluated after the arithmetic operators but before the logical operators.

The Relational Operators are:

| | |
|---|---|
| = | Equal |
| <> | Not Equal |
| < | Less Than |
| > | Greater Than |
| <= | Less Than or Equal |
| >= | Greater Than or Equal |

A relational expression has a value of -1 if it is evaluated to be true and zero if it is evaluated to be false. For example:

5+6*5>15*2 is true.

Relational operators can be used to perform comparisons between two strings for example, whether A$=B$.

In performing string comparisons, the system does a left-to-right comparison. This is based on the ASCII collating sequence of the numeric codes in the characters of the strings being compared (including such characters as leading and trailing spaces).

Examples: Ex. 1

```
170  IF A<B+3 THEN 160
180  IF A=B+3 THEN PRINT "A HAS THE VALUE "
190  IF A>=B THEN T1=B
200  IF A$=B$ THEN PRINT "EQUAL ":A=1/B
210  IF A>B THEN PRINT "GREATER " ELSE PRINT
"NOT GREATER"
220  IF X THEN Y=X
230  IF X=0 GOTO 1600
```

Ex. 2

```
TRACE¶
10   REM IF...THEN...ELSE EXAMPLE¶
15   ;¶
20   INPUT "F="F¶
40   C=(F-32)*5/9¶
50   IF F>=0 AND F<=32 THEN 70¶
60   IF F>=212 THEN 165 ELSE 100¶
70   REM PATH TAKEN FOR F=0 TO 32¶
80   REM¶
90   REM¶
100  ; "F=" F,"C=" C ! PATH TAKEN FOR F>32 TO <212¶
110  GOTO 15¶
165  REM PATH TAKEN FOR F >= 212¶
170  ;  "END OF TEST"¶
180  END¶
RUN¶

10   15
20   F=-30¶  (USER ENTERS -30)
40   50 60 100 F=-30  C=-34.444
110  15
20   F=21¶  (USER ENTERS 21)
40   50  70  80  90  100 F=21      C=-6.11111
110  15
20   F=38¶  (USER ENTERS 38)
40   50 60 100 F=38  C=3.33333
110  15
20   F=400¶  (USER ENTERS 400)
40   50 60 165 170 END OF TEST
180
BASIC
```

## 9.12 NEXT STATEMENT

Function: Terminates a program loop which began with a FOR statement.

Mode: Program

Format: NEXT <variable>

Arguments: Variable is the same variable specified in the FOR statement. Together the FOR and NEXT statements describe the boundaries of the program loop.

Use: When execution encounters the NEXT statement for positive step values, the computer adds the STEP expression value to the variable. When the condition for execution of the FOR statement is no longer true, it checks to see if the variable is still less than or equal to the terminal expression value. When the variable exceeds the terminal expression value, control falls through the loop to the statement following the NEXT statement.

If the step value is negative and the expression exceeds the variable, the next statement executed is the one following NEXT; otherwise, the program will execute statements between FOR and NEXT and repeat the test.See FOR statement, Section 9.8.

Example: See FOR statement, Section 9.8.

## 9.13 NOTRACE STATEMENT

Function: Terminates the printout of line numbers initiated by TRACE statement.

Mode: Direct/Program

Format: NOTRACE

Example:

```
10   PRINT "BEGIN "¶
20   K=-1¶
30   TRACE¶
40   IF K>1 THEN 80¶
50   K=K+1¶
60   PRINT "NUMBER ";K¶
70   GOTO 40¶
80   A=K¶
90   NO TRACE¶
100  PRINT "STOP"¶
RUN¶
BEGIN
40 50 60 NUMBER 0
70 40 50 60 NUMBER 1
70 40 50 60 NUMBER 2
70 40 80 90
STOP
```

The TRACE function is disabled before line 40 and after line 90.

## 9.14 ON ERROR GOTO STATEMENT

Function: Specifies a user routine for error handling.

Mode: Program

Format: ON ERROR GOTO [line no]

Arguments: The specified line no. is the start of an error routine.

Use: Normally the occurrence of an error causes termination of the user program execution and the printing of a diagnostic message.

Some applications may require the continued execution of a user program after an error occurs. In these situations, the user can execute an ON ERROR GOTO statement within the program.

This statement is placed in the program prior to any executable statements with which the error handling routine deals. The system will then know that a routine exists that will take over and analyze any I/O or computional error encountered in the program and possibly make an attempt to recover from that error.

The variable ERRCODE is associated with the statement and available for the user program.

For Error Codes and Messages see Appendix B.

If there are portions of the user program in which any errors detected are to be processed by the system and

not by the user program, the error routine can be disabled by:

```
line no  ON ERROR GOTO
```

without a line number following GOTO, which returns control of error handling to the system.

Note: For proper operation of the ON ERROR GOTO <line no.> statement, the error handling routine must execute a RESUME.

Example:

```
10 REM THIS PROGRAM ACCEPTS ONLY POSITIVE NUMBERS.¶
20 ON ERROR GOTO 80¶
30 REM "CON:" IS OPEN AS FILE 0%¶
40 INPUT "POSITIVE NUMBER"A¶
50 Z=SQR(A)¶
60 PRINT "SQUARE ROOT OF:" A "IS----->" Z¶
70 STOP¶
80 FOR I=1 TO 10¶
85   ; CHR$(7) ! SYSTEM BEEPS¶
87 NEXT I¶
90 PRINT "ENTRY ERROR----ONLY POSITIVE NUMBERS"
95 PRINT "ALLOWED"¶
100 RESUME¶
110 END¶
RUN¶
POSITIVE NUMBER? 25
SQUARE ROOT OF 25 IS -----> 5
STOP IN LINE 70
BASIC¶
RUN¶
POSITIVE NUMBER? -10
(system beeps 10 times)
ENTRY ERROR----ONLY POSITIVE NUMBERS ALLOWED
BASIC
```

## 9.15 ON...GOSUB...STATEMENT

**Function:** Conditionally transfers control to one of several subroutines or to one of several entry points in one subroutine.

**Mode:** Program

**Format:** ON <expression> GOSUB <line no. 1> [,line no. 2,...]

**Arguments:** Depending on the rounded integer value of the expression, control is transferred to the subroutine which begins at one of the line numbers listed. When the subroutine returns, execution is resumed at the statement following the ON GOSUB statement. If the value of the expression addresses a line number outside the range of the list, an error message will be displayed.

**Use:** Since it is possible to transfer control into a subroutine at different points, the ON - GOSUB statement could be used to determine which part of the subroutine should be executed. An error message will be generated if outside the range. See also ON ... GOTO statement.

**Example:**

```
10    FOR X = 1.7 to 5.9 STEP .6¶
20    PRINT X;¶
40    ON X GOSUB 1300, 200, 1300, 400, 1300, 1300¶
50    PRINT A$¶
60    NEXT X¶
70    GOTO 9999¶
200   LET A$ = "SUB200"¶
210   RETURN¶
400   LET A$ = "SUB400"¶
410   RETURN¶
1300  LET A$ = "SUB1300"¶
1310  RETURN¶
9999  END¶
```

Control is transferred to:

```
line  200 for X = 1.7
       200      2.3
       200      2.9
      1300      3.5
       400      4.1
      1300      4.7
      1300      5.3
      1300      5.9

RUN¶
1.7   SUB200
2.3   SUB200
2.9   SUB1300
3.5   SUB1300
4.1   SUB1400
4.7   SUB1300
5.3   SUB1300
5.9   SUB1300
BASIC
```

## 9.16 ON...GOTO STATEMENT

Function: Transfers control to one of several lines depending on the value of the expression at the time the statement is executed.

Mode: Program

Format: ON <expression> GOTO <line no.1>[,line no.2,...,...]

Arguments: Expression can be any legal arithmetic expression.

Line no. is where control is transferred to as illustrated in the example below.

Use: ON...GOTO permits the program to respond to multiple choices. It eliminates the necessity of separate IF statements for each alternative. The expression is evaluated and rounded to the nearest integer. This integer is used as an index or as a pointer to one of the line numbers in the list. An error message will be generated if it is outside the range.

Example:

```
100 ON A/B GOTO 1000,1500,1700
```

transfers control to:

1. line number 1000 if .5 < = A/B < 1.5
2. line number 1500 if 1.5 < = A/B < 2.5
3. line number 1700 if 2.5 < = A/B < 3.5
4. gives error if A/B < 0.5
5. gives error if A/B $\geq$ 3.5

## 9.17 ON...RESTORE STATEMENT

Function: Restores the DATA-pointer by the same selection routine as the ON-GOTO statement.

Mode: Program

Format: ON <expression> RESTORE <line no.1>[,line no.2,...,...]

Arguments: Expression can be any legal arithmetic expression.

Line no. is where the DATA-pointer is restored to as explained below.

Use: This statement can be used to reset the DATA-pointer to a specific point in the data buffer. The expression is evaluated and rounded to the nearest integer. This integer is used as an index to set the DATA-pointer to the corresponding list number. An error message will be generated if it is outside the range.

Example:

```
AUTO¶
10  FOR X=1 TO 3¶
20  READ  A,B,C¶
30  ON X RESTORE 60,70,80¶
40  PRINT A,B,C¶
50  NEXT X¶
60  DATA 1,2,3¶
70  DATA 4,5,6¶
80  DATA 7,8,9¶
90  END¶
RUN¶
 1 2 3
 1 2 3
 4 5 6
BASIC
```

## 9.18 ON...RESUME STATEMENT

Function: Transfers control to one of several line numbers depending on the value of the expression in error handling situations.

Mode: Program

Format: ON <expression> RESUME <line no.1>[,line no.2,...,...]

Arguments: Expression can be any legal arithmetic expression.

Use: This statement is used to accomplish a conditional return from an error handling routine. The expression is evaluated and rounded to the nearest integer. This integer is used as an index to one of the line numbers in the list. ON...RESUME is used with ON ERROR GOTO as described in Section 2.12.

Example:

```
10   ON ERROR GOTO 100
.
.
.
100  REM ERROR HANDLER
.
.
.
150  ON B RESUME 1000,2000
.
.
.
```

## 9.19 PAUSE STATEMENT

Function: Pauses the current BASIC program task.

Mode: Direct/Program

Format: PAUSE

Use: The PAUSE statement allows the user to temporarily suspend the current BASIC program task, in order to communicate with the operating system or another task, without losing the current BASIC program workspace. The PAUSE statement should be used (rather than CTRL-A) to invoke and communicate with another task if that task requires interactive console input. The PAUSE statement is identical in function to the monitor PAUSE command, but in addition, also closes the BASICERR file, if it is open; other files are not affected. The PAUSE statement allows the system disk to be changed; see Section 2.14 for additional information.

The paused BASIC program task can be resumed by using the operating system CONTINUE command or the CONT key. If the BASIC task was invoked by typing "BASIC" (rather than using the RUN or START commands), then the task (USPO) can be resumed by simply typing the CONT key in response to the monitor prompt. If the BASIC program was given a different task name, then the task can be resumed simply by typing in the task name following the monitor prompt and terminating it with the CONT key, or by using the CONTINUE command (see the 8800 Series Utility Programs Programmer's Reference Manual for additional information).

Example: See Section 2.14.

## 9.20 RESUME STATEMENT

**Function:** Transfers control to a specified line number from an error subroutine or to the statement which caused the error.

**Mode:** Program

**Format:** RESUME [line no.]

**Arguments:** Line no. specifies where program execution will continue. If it is omitted, program execution continues at the beginning of the statement which caused the error.

**Example:**

```
LIST¶
10  REM THIS PROGRAM WORKS FOR ONLY POSITIVE NUMBERS.
15  REM -----FUNCTIONALITY OF RESUME-----
20  ON ERROR GOTO 80
30  REM "CON:" IS ALWAYS OEPN AS FILE 0%
40  INPUT "POSITIVE NUMBER" A
50  Z=SQR(A)
60  PRINT "SQUARE ROOT OF:" A "IS-------->" Z
70  STOP
80  ; "EINSTEIN-----ONLY POSITIVE NUMBERS ALLOWED"
85  ; CHR$(7)
90  RESUME 40
100 END
BASIC
```

## 9.21 RETURN STATEMENT

Function: Transfers control back to the statement after the calling GOSUB or causes a return from a multiple line function.

Mode: Program

Format:
1. RETURN
2. RETURN <expression>

Argument: Expression is any valid Monroe BASIC expression containing constants and variables.

Use: Format 1 is used to transfer control back to the statement following the original GOSUB statement. After having reached the subroutine through a GOSUB or an ON...GOSUB statement, the subroutine is executed until the interpreter encounters a RETURN statement. Subroutines can be nested, that is one subroutine can call another subroutine or itself.

Format 2 is used when RETURN is part of a multiple line DEF function. When this RETURN statement is encountered, the expression is evaluated and used as the value of the DEF function. An exit is then performed from the definition. The DEF can contain more than one RETURN statement.

Examples:

Ex. 1

```
LIST¶
50  GOSUB 1300
.
.
.
1300  REM ** SUBROUTINE 1***
1400  LET K=1
.
.
.
2000  RETURN
.
.
.
9999  END
BASIC
```

Ex. 2

```
AUTO¶
10  DEF FNM (X,Y)¶
20    IF Y<X THEN RETURN X¶
30    RETURN Y¶
40  FNEND¶
50  ¶
BASIC
```

## 9.22 STOP STATEMENT

Function: Terminates program execution.

Mode: Program

Format: STOP

Use: The STOP statement terminates the execution of the program. The variables are not reset and the open files remain open. Program execution can be continued by one of these commands: CON or GOTO.

The STOP statement differs from the END statement in that it causes Monroe BASIC to display the statement number where the program halted. It can occur several times in a single program and is recommended for debugging purposes.

Example:

```
|
|
|
100 STOP
```

The message displayed is:

STOP IN LINE line number

## 9.23 TRACE STATEMENT

**Function:** Prints the line numbers of the executed program lines.

**Mode:** Direct/Program

**Format:** TRACE [#channel no.]

**Argument:** Channel no. is the internal file number representing the destination where trace data is to be sent. The default is the user's console.

**Use:** TRACE is used when debugging a program to track the execution of the program.

**Example:**

```
LIST¶
100  OPEN "PR:" AS FILE 1%
110  A=15.345
115  TRACE #1%
120  B=153¶
125  IF A=0 THEN STOP
130  C%=B
135  X=A*2
140  D1%=A
145  NO TRACE
150  PRINT #1% A,B,C%,D1%,X
160  CLOSE 1%
170  END
RUN¶
```

(The following text will be printed on the printer when the above program is executed:)

```
120 125 130 135 140 145
15.345 153 153 15 30.69
```

## 9.24 WEND STATEMENT

Function: Defines the limit of the WHILE loop.

Mode: Program

Format: WEND

Use: When the WEND statement is executed, control is transferred to the last non-terminated WHILE statement.

Example: See WHILE statement in this section.

## 9.25 WHILE STATEMENT

Function: Defines a specific condition for leaving a loop.

Mode: Program

Format: WHILE <relational expression>

Argument: Relational expression is some test condition.

Use: WHILE should be used only in iterative loops where the logical loop structure modifies the values that determine the loop termination. This is a significant departure from FOR loops in which control is automatically iterated.

There are many situations in which the final value of the loop variable is unknown in advance. What is desired is to execute the loop as many times as necessary to satisfy some special conditions specified by WHILE.

Examples:

```
10  WHILE X < 10
20  X = X*X + 1
30  WEND
```

Before the loop is executed and at each loop iteration the condition X < 10 is tested. The iteration continues if the result is true.

The above example is equivalent to:

```
10  IF X > = 10 THEN 40
20  X = X*X + 1
30  GOTO 10
40  ............
```

# SECTION 10
# FUNCTIONS

# SECTION 10
# FUNCTIONS

## 10.1 INTRODUCTION

Functions in the context of Monroe BASIC are independent programs stored in the interpreter which perform specific mathematical, string, or miscellaneous operations. A user program can include a call to a Monroe BASIC function program whenever it requires the execution of any of these operations. These functions can save a great deal of coding time. They enable the user to include the function without having to know the details behind them.

This section discusses three type of functions:

1. Mathematical
2. String
3. Miscellaneous

## 10.2 MATHEMATICAL FUNCTIONS

When programming, the user may encounter many cases where relatively common mathematical operations are performed. The results of these common operations are likely to be found in mathematical tables; i.e., sine, cosine, square root, log, etc. Since the computer can perform this type of operation with speed and accuracy, these operations are built into Monroe BASIC. Internal functions can be called whenever such a value is needed. For example:

```
SIN (23.*PI/180.)
LOG (144.)
```

The various mathematical functions available are listed in Table 10-1.

Table 10-1. Mathematical Functions

| Function | Result |
|---|---|
| ABS(x) | Returns absolute value of x. |
| ATN(x) | Returns arctangent of x in radians. |
| COS(x) | Returns cosine of x in radians. |
| EXP(x) | Returns exponential function (i.e., $e^x$). |
| FIX(x) | Returns the truncated value of x. |
| HEX$(x) | Returns the hexadecimal string representation of a decimal number. |
| INT(x) | Returns the greatest integer that is less than or equal to x. |

| Function | Result |
|---|---|
| LOG(x) | Returns the natural logarithm of x (i.e., log $e^x$). |
| LOG10(x) | Returns the common logarithm (base 10) of x. |
| MOD(x,y) | Returns the remainder of the integer division x/y. |
| OCT$(x) | Returns the octal string representation of a decimal number. |
| PI | Returns a constant value of 3.1415927. |
| RND | Returns a random number between 0 and 0.999999. |
| SGN(x) | Returns the sign of x. |
| SIN(x) | Returns the sine of x. |
| SQR(x) | Returns the square root of x. |
| TAN(x) | Returns the tangent of x. |

Each function listed in Table 10-1 is described in detail in subsequent paragraphs.

Order of Execution

A mathematical function is executed in the following manner:

1. The operation or operations within the argument are performed.
2. The function itself is evaluated.
3. The remaining arithmetic operations in the statement are performed in their normal order or precedence.

## ABS Function

**Function:** Returns the absolute value of x

**Mode:** Direct/Program

**Format:** ABS(x)

**Argument:** x is numerical

**Result:** Floating

**Example:**

```
; ABS(- 123)¶
  123
BASIC
```

## ATN Function

**Function:** Returns the arctangent of x.

**Mode:** Direct/Program

**Format:** ATN(x)

**Argument:** x is in radians

**Result:** Floating

**Example:**

```
; ATN(5)¶
  1.3734
BASIC
```

## COS Function

**Function:** Returns the cosine of x.

**Mode:** Direct/Program

**Format:** COS(x)

**Argument:** x is in radians

**Result:** Floating

**Example:**

```
10 A = .57¶
20 B = COS (A)¶
30 PRINT B¶
40 END¶

RUN¶
 .841901
BASIC
```

## EXP Function

**Function:** Returns the value of the antilog ($e^x$ where e = 2.71828, single precision).

**Mode:** Direct/Program

**Format:** EXP(x)

**Argument:** $-88 \leq x \geq 88$

**Result:** Floating

**Example:**

```
PRINT EXP (1)¶
 2.71828
BASIC
```

## FIX Function

**Function:** Returns the truncated value of x.

**Mode:** Direct/Program

**Format:** FIX(x)

**Argument:** x is numeric

**Result:** Floating

**Example:**

```
PRINT FIX (-123.96)¶
-123
BASIC
```

## HEX$ Function

**Function:** Converts a decimal number into a hexadecimal string.

**Mode:** Direct/Program

**Format:** HEX$(x)

**Argument:** x is decimal number

**Result:** String

**Example:**

```
10 Y$=HEX$(255)¶
20 ; Y$¶
RUN¶
FF
BASIC
```

## INT Function

**Function:** Returns the greatest integer which is less than or equal to X.

**Mode:** Direct/Program

**Format:** INT(x)

**Argument:** x is numeric

**Result:** Floating

**Use:** The integer function returns the value of the greatest integer not greater than x.

INT can also be used to round to any given decimal place, by asking for:

```
INT(X*10.**D%+.5)/10.**D%
```

Where D% is the number of decimal places desired.

If the number is negative, INT will return the largest integer less than the argument.

**Examples:**

Ex. 1

```
10 Y=INT(34.67)
```

The result is Y=34

Ex. 2

```
10 Y=INT(34.67+.5)
```

The result is Y=35

Ex. 3

```
10 Y=INT(-23.15)
```

The result is Y=-24

Ex. 4

```
1200  INPUT "NUMBER TO BE PROCESSED BY INT", A¶
1210  INPUT "NUMBER OF DEC. PLACES FOR ROUNDING", D¶
1220  PRINT "TRUNCATED INTEGER=";INT(A)¶
1230  PRINT "ROUNDED INTEGER=";INT(A+.5)¶
1240  PRINT "ROUNDED TO ";D; "PLACES=";¶
1250  PRINT INT(A*10**D+.5)/(10**D)¶
1300  PRINT¶
1310  PRINT "ENTER ANOTHER NUMBER, TYPE A ZERO TO STOP"¶
1320  INPUT A¶
1330  IF A < > 0 THEN GO TO 1210¶
9999  END¶

RUN¶
 NUMBER TO BE PROCESSED BY INT? 13.56¶
 NUMBER OF DEC. PLACES FOR ROUNDING? 1¶
 TRUNCATED INTEGER=13
 ROUNDED INTEGER=14
 ROUNDED TO 1 PLACES=13.6

 ENTER ANOTHER NUMBER, TYPE A ZERO TO STOP
 ? 123.4567¶
 NUMBER OF DECIMAL PLACES FOR ROUNDING? 2¶
 TRUNCATED INTEGER=123
 ROUNDED INTEGER=123
 ROUNDED TO 2 PLACES=123.46

ENTER ANOTHER NUMBER, TYPE A ZERO TO STOP
? 0¶
BASIC
```

## LOG Function

**Function:** Returns the natural logarithm of X, $\log_e X$.

**Mode:** Direct/Program

**Format:** LOG(x)

**Argument:** x > zero

**Result:** Floating

**Example:**

```
PRINT LOG (2)¶
0.693147
BASIC
```

## LOG10 Function

**Function:** Returns the common logarithm of x, $\log_{10}x$.

**Mode:** Direct/Program

**Format:** LOG10(x)

**Argument:** x > zero

**Result:** Floating

**Example:**

```
10 A = LOG10(5)¶
20 PRINT 2*A¶
30 END¶
RUN¶
 1.39794
BASIC
```

MOD Function

Function: Returns an integer value representing the remainder of a division of the arguments.

Mode: Direct/Program

Format: MOD(x,y)

Argument: x and y are numeric.

Result: Integer

Example:
```
; MOD(22,4)¶
 2
BASIC
```

## OCT$ Function

**Function:** Converts a decimal number into an octal string.

**Mode:** Direct/Program

**Format:** OCT$(x)

**Argument:** x is a decimal number

**Result:** String

**Example:**

```
; OCT$(59)¶
73
BASIC
Y$=OCT$(59)¶
;Y$¶
73
BASIC
```

## PI Function

**Function:** Returns a constant value of 3.14159 (single precision) or 3.141592653589793 (double precision).

**Mode:** Direct/Program

**Format:** PI

**Result:** Floating

**Example:**

```
10 INPUT R¶
20 C = 2*PI*R¶
30 PRINT C¶
40 END¶
RUN¶
? 11¶
 69.115
BASIC
```

RND Function

Function: Returns a random number between 0 and 0.999999.

Mode: Direct/Program

Format: RND

Use: RND is used to return a random number between 0 and 0.999999. The function will generate the same random number sequence every time the program is run unless a RANDOMIZE statement is placed before RND in the program.

Result: Floating

Example: Ex. 1

```
10 Y=RND
```

Ex. 2

```
10 Y=(D-A)*RND+A
```

Y will be assigned a random number between A and D.

SGN Function

Function: Returns the sign of X.

Mode: Direct/Program

Format: SGN(x)

Argument: x is numeric.

Result: Integer

Use: The sign function returns a value of +1 if X is a positive value, 0 if X is 0, and -1 if X is negative. For example: SGN(3.42) = 1, SGN(-42) = -1, and SGN(23-23) = 0.

Example:

```
1000   REM - SGN FUNCTION DEMO¶
1010   READ A, B¶
1100   PRINT "A=";A, "B=";B¶
1110   PRINT "SGN(A)="; SGN(A), "SGN(B)=" SGN(B)¶
1120   PRINT "SGN(INT(A))=";SGN(INT(A))¶
1200   DATA -5.43, 0.21¶
9999   END¶

RUN¶
 A=-5.43          B=.21
 SGN(A)=-1        SGN(B)=1
 SGN(INT(A))=-1
BASIC
```

## SIN Function

**Function:** Returns the sine of x.

**Mode:** Direct/Program

**Format:** SIN(x)

**Argument:** x is in radians.

**Result:** Floating

**Example:**

```
PRINT SIN (.57)¶
 .539632
BASIC
PRINT SIN (PI/2)¶
 1
BASIC
```

## SQR Function

**Function:** Returns the square root of x.

**Mode:** Direct/Program

**Format:** SQR(x)

**Argument:** x $\geq$ zero

**Result:** Floating

**Example:**

```
;SQR(9)¶
 3
BASIC
```

## TAN Function

**Function:** Returns the tangent of x.

**Mode:** Direct/Program

**Format:** TAN(x)

**Argument:** x is in radians.

**Result:** Floating

**Example:**

```
10 INPUT A¶
20 PRINT "SIN(A)/COS(A)=";SIN(A)/COS(A)¶
30 ;"TAN(A)=";TAN(A)¶
40 END¶
RUN¶
?0.57¶
 SIN(A)/COS(A)= .640969
 TAN(A)= .640969
BASIC
```

## 10.3 STRING FUNCTIONS

Besides intrinsic mathematical functions (e.g., SIN, LOG), various functions for use with character strings are provided. These functions allow the program to perform arithmetic operations with numeric strings, concatenate two strings, access a part of a string, determine the number of characters in a string, and perform other useful operations. These functions are particularly useful when dealing with whole lines of alphanumeric information input by an INPUT LINE statement. The various string functions available are summarized in Table 10-2.

Table 10-2. String Functions

| Function | Description |
|---|---|
| ADD$ | Returns the result of adding two numeric strings. |
| ASCII or ASC | Returns the ASCII decimal value for the first character in a string. |
| A$+B$ | Returns the concatenation of two strings. |
| CHR$ | Returns a character-string having the ASCII value of arguments. |
| COMP% | Returns a truth value based on result of numeric comparison. |
| DIV$ | Returns a quotient. |
| INSTR | Searches for and returns the location of a substring within a string. |
| LEFT$ | Returns left substring of an existing string. |
| LEN | Returns the length of a string. |

Table 10-2. String Functions (Cont.)

| Function | Description |
|---|---|
| MID$ | Returns or replaces a substring of a string. |
| MUL$ | Returns the result of multiplying two numeric strings. |
| NUM$ | Returns a string of numeric characters. |
| RIGHT$ | Returns a right substring of a string. |
| SPACE$ | Returns a string of spaces. |
| STRING$ | Creates and returns a string of ASCII characters. |
| SUB$ | Returns the result of subtracting two numeric strings. |
| VAL | Returns the numeric value of the string of numeric characters. |

Each string function is described in detail in subsequent paragraphs.

## ADD$ Function

**Function:** Adds the values of two numeric strings to a specified number of decimal places.

**Mode:** Direct/Program

**Format:** ADD$(A$,B$,p%)

**Argument:** A$ and B$ are numeric strings.
p% when positive specifies the number of decimals in the result and when negative specifies the number of places of precision desired.

**Result:** String

**Note:** ASCII arithmetic calculations can operate on up to 125 characters.

**Example:**

```
S$="12349.178"¶
BASIC
PRINT ADD$(S$,"89.454",3)¶
 12438.632
BASIC
```

## ASCII Function

**Function:** Returns an integer equal to the ASCII value of the first character of a string.

**Mode:** Program/direct

**Format:**
1. ASCII(string) or
2. ASC(string)

**Note:** If the second form ASC is entered, it will be listed as the first form, ASCII.

**Argument:** String is a string expression.

**Result:** Integer

**Example:**

```
; ASCII("T")¶
 84
BASIC
10 A$="XAB"¶
20 ;ASCII(A$)¶
RUN¶
 88
BASIC
```

**Note:** The returned value is zero if A$ is null.

## CHR$ Function

**Function:** Returns a character-string corresponding to the ASCII value of the arguments.

**Mode:** Direct/Program

**Format:** CHR$(n1[,n2,n3...])

**Argument:** n is the ASCII decimal of the character desired.

**Result:** String

**Example:**

```
A$=CHR$(65,66,67)¶
BASIC
;A$¶
 ABC
BASIC
```

## COMP% Function

**Function:** Returns a truth value based on the result of a numeric comparison.

**Mode:** Direct/Program

**Format:** COMP%(A$,B$)

**Argument:** A$ and B$ are numeric strings.

**Result:** Integer

**Use:** The truth values are as follows:

```
-1 IF A$ < B$
 0 IF A$ = B$
 1 IF A$ > B$
```

**Example:**

```
A$="12345.6789":B$="9876.54321"¶
BASIC
T%=COMP% (A$,B$)¶
BASIC
PRINT T%¶
 1
BASIC
PRINT COMP% (B$, A$)¶
 -1
BASIC
```

## DIV$ Function

**Function:** Returns a quotient, A$ divided by B$.

**Mode:** Direct/Program

**Format:** DIV$(A$,B$,p%)

**Argument:** A$ and B$ are numeric strings.
A$ is the numerator and B$ is the denominator.
p% when positive is the number of decimal places in the quotient and when negative specifies the number of places of precision desired.

**Result:** String

**Note:** ASCII arithmetic calculations can operate on up to 125 characters.

**Example:**

```
10 C$="3.5"¶
20 V9$=DIV$(C$,"1.7777",3%)¶
30 PRINT V9$¶
40 END¶
RUN¶
 1.969
BASIC
```

## INSTR Function

**Function:** Searches for and returns the location of a substring within a string.

**Mode:** Direct/Program

**Format:** INSTR(n%,A$,B$)

**Argument:** A$ is a string.
B$ is the substring within A$ you want to locate.
n% is the character position within A$ where the search will begin.

**Result:** Integer

**Use:** A value of 0 is returned if B$ is not in A$ or the character position of B$ if B$ is found to be in A$ (character position is measured from the start of the string with the first character counted as character 1).

**Example:**

```
A$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"¶
BASIC
PRINT INSTR(5%,A$,"OP")¶
 15
BASIC
```

## LEFT$ Function

**Function:** Returns a substring of an existing string.

**Mode:** Direct/Program

**Format:** LEFT[$](A$,n)

**Argument:** A$ is a string.
n is character position in A$ where the substring will end. N = 0 is permitted. N must be $\leq$ the length of A$.

**Result:** String

**Use:** The substring will begin with the first character in A$ and end with the nth character.

**Example:**

```
10  A$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"¶
20  ;LEFT$(A$,6)¶
30  END¶
RUN¶
     ABCDEF
BASIC
```

## LEN Function

**Function:** Returns the length of a string.

**Mode:** Direct/Program

**Format:** LEN(A$)

**Argument:** A$ is a string.

**Result:** Integer

**Example:**

```
PRINT LEN ("JOHN SMITH")¶
 10
BASIC
```

## MID$ Function

**Function:** Returns or replaces a substring of a string.

**Mode:** Direct/Program

**Format:**
MID[$](A$,n1,n2)
[LET] MID[$](A$,n1,n2)=<string expression>

**Argument:**
A$ is a string.
n1 is the character position in A$ where the substring begins.
n2 is the number of characters in the substring.
n2 = 0 is permitted.
n1 + n2 must not exceed one more than the string length.

**Result:** String

**Note:** This function can also be used on the left-hand side of a LET statement. The length of the string on the right hand side must be of length n2.

**Use:** The characters between and including n1 through n1+n2-1 characters of A$ comprise the substring.

**Example:**

```
10 ;"NAME,ADDRESS? ";¶
20 INPUTLINE A$¶
30 PRINT¶
40 Z=INSTR(1,A$,",")¶
50 Y=LEN(A$)¶
60 ;"NAME= ";LEFT$(A$,Z-1)¶
70 ;"ADDRESS= ";MID$(A$,Z+1,Y-(Z+1))¶
80 B$= "MONROE          FOR BUSINESS"
90 MID$ (B$,8,7)="SYSTEMS"
100 ;B$
110 END¶
RUN¶
NAME,ADDRESS? MONROE,USA¶
NAME= MONROE
ADDRESS= USA
MONROE SYSTEMS FOR BUSINESS
BASIC
```

## MUL$ Function

**Function:** Returns the result of multiplying two numeric strings.

**Mode:** Direct/Program

**Format:** MUL$(A$,B$,p%)

**Argument:** A$ and B$ are numeric strings.
p when positive specifies the number of decimal places required and when negative, the number of places of precision desired.

**Result:** String

**Example:**

```
10 INPUT A$,B$¶
20 ;MUL$(A$,B$,6)¶
30 END¶
RUN¶
?12345.6789,987.54321¶
 12191891.370535
BASIC
```

## NUM$ Function

**Function:** Returns a string of numeric characters representing the value of n as it would be displayed by a PRINT statement.

**Mode:** Direct/Program

**Format:** NUM$(n)

**Argument:** n is a string of numeric characters.

**Result:** String

**Example:**

```
;NUM$(123456789012)¶
1.234568E+11
BASIC
```

**Note:** Returned string will not have any leading blanks.

## RIGHT$ Function

**Function:** Returns a particular substring of a string.

**Mode:** Direct/Program

**Format;** RIGHT[$](A$,n)

**Argument:** A$ is a string.
n is the character position in A$ where the substring will begin. n can equal LEN(A$) + 1 which results in an empty string, but cannot be greater than A$ + 1.

**Result:** String

**Use:** RIGHT$ returns the characters from the nth character through the last character in A$.

**Example:**

```
10 ;"NAME,ADDRESS? ";¶
20 INPUT LINE A$¶
30 PRINT¶
40 Z=INSTR(1,A$,",")¶
50 ;"NAME= ";LEFT$(A$,Z-1)¶
60 ;"ADDRESS= ";RIGHT$(A$,Z+1)¶
70 END¶
RUN¶
NAME,ADDRESS? MONROE, USA¶
 NAME= MONROE
 ADDRESS= USA
BASIC
```

## SPACE$ Function

**Function:** Returns a string of a specified number of spaces.

**Mode:** Direct/Program

**Format:** SPACE$(N%)

**Argument:** N% is the number of spaces.

**Result:** String

**Example:**

```
PRINT "ABC";SPACE$(10);"DEF"¶
 ABC          DEF
BASIC
```

## STRING$ Function

**Function:** Returns a string of ASCII characters.

**Mode:** Direct/Program

**Format:** STRING$(n1,n2)

**Argument:** n1 is the length of the string in characters.
n2 is the ASCII decimal value of the character.

**Result:** String

**Example:** Print a string of 15 *'s.

```
10 F$=STRING$(15,42)¶
20 PRINT F$¶
30 END¶
RUN¶
***************
BASIC
```

## SUB$ Function

**Function:** Subtracts two numeric strings and gives the result to a specified number of decimal places.

**Mode:** Direct/Program

**Format:** SUB$(A$,B$,p%)

**Argument:** A$ and B$ are numeric strings.
p% when positive specifies the number of decimal places in the result and when negative, the number of places of precision desired.

**Result:** String

**Note:** ASCII arithmetic calculations can operate on up to 125 characters.

**Example:**

```
10 B$="9876.54321"¶
20 ;SUB$(B$,"98.76",5)¶
30 END¶
RUN¶
9777.78321
BASIC
```

## VAL Function

**Function:** Computes and returns the numeric value of a string of numeric characters.

**Mode:** Direct/Program

**Format:** VAL(<string>)

**Argument:** String is a numeric string. The result is a floating point number. If the string contains non-numeric characters other than +, -, or E for exponential, an error is generated.

**Result:** Floating

**Example:**

```
10  A=VAL("14.3E-5")¶
20  PRINT A¶
30  END¶
RUN¶
.000143
BASIC
```

## 10.4 MISCELLANEOUS FUNCTIONS & STATEMENTS

The following are described in this section:

| Item | Use |
|---|---|
| CUR | Positions the cursor on specified line and column. |
| CURREAD | Reads the current cursor position. |
| ERRCODE | Returns the value of the most recent error code. |
| FN | Calls a user-defined function. |
| PDL | Returns a specific joystick's X or Y coordinate or a value specifying which joystick and control button was depressed. |
| REM or ! | Insert comments into a user's program. |
| SLEEP | Stops the running of a program for a specified number of seconds. |
| SOUND | Returns sounds on system speakers with specified qualities. |
| TAB | Tabulates to the specified column on the line. |
| TIME$ | Returns year-month-day, hour.minutes.seconds. |

## CUR Function

**Function:** Moves the cursor to the specified row and column on the screen.

**Mode:** Program/Direct

**Format:** CUR(y%,x%)

**Argument:** y% is the line where the cursor is to be moved with values of 0 to 23.

x% is the column position on the line with values of 0 to 39/79 (39 is valid for education model and 79 for business model).

**Result:** String

**Use:** This function generates a string which, when printed, places the cursor at the specified row and column on the screen.

**Example:**

Ex. 1

```
100  PRINT CUR(12,20) "1980 STATUS REPORT"¶
200  ; CUR(13,22) "EASTERN DIVISION"¶
300  . . .
.
.
.
```

Ex. 2

```
10 PRINT CUR (10,10); "COLUMN 10, ROWN 10"¶
20 A$ = CUR (1,2) + 'ROW 1, COLUMN 2' : PRINT A$¶
```

## ERRCODE Function

**Function:** Returns the value of the latest generated error code.

**Mode:** Direct/Program

**Format:** ERRCODE

**Use:** The ERRCODE function is normally used in conjunction with the IF and ON statements. If no error has been indicated the function value is 0.

**Result:** Integer

**Example:**

```
LIST¶
10  REM THIS PROGRAM WORKS FOR POSITIVE NUMBERS ONLY.
20  REM-----
30  ON ERROR GOTO 90
40  OPEN "CON:" AS FILE 0%
50  INPUT "POSITIVE NUMBER "A;
60  Z=SQR(A)
70  PRINT "SQUARE-ROOT OF:" A "IS-->" Z
80  STOP
90  IF ERRCODE=142 THEN ; " NO NEGATIVE NUMBERS ALLOWED"
100 IF ERRCODE=210 THEN ; " NO CHARACTERS ALLOWED"
110 ; ERRCODE
120 ; CHR$(7); : REM BELL SOUNDS AFTER ERROR MESSAGE
125 REM IS PRINTED
130 RESUME 50
140 END
RUN¶
 POSITIVE NUMBER? A¶ NO CHARACTERS ALLOWED
 210 (bell sounds)
 POSITIVE NUMBER? -5¶ NO NEGATIVE NUMBERS ALLOWED
 142 (bell sounds)
 POSITIVE NUMBER 9¶ SQUARE-ROOT of: 9 IS--> 3
 .
 .
 .
```

## FN Function

**Function:** Calls a user-defined function.

**Mode:** Direct/Program

**Format:** FN<name> [<type>] [(arguments)]

**Arguments:** Name is any valid variable name.

Type is optional and can be either % (or ".") or $.

Parameter consists of one or more expressions which are passed to the defined function. They must be specified if they were included in the DEF FN statement.

**Result:** Depends on the type.

**Use:** This function allows the programmer to call a user-defined function in the same way as, for example, SIN would be called.

**Note:** Refer to DEF FN statement, Section 9 for how to define the function.

**Examples:**

Ex. 1

```
EXTEND¶
BASIC
LIST¶
5   REM **DEFINE AND USE SECANT FUNCTION***
10  DEF FNSEC(X)=1/SIN(X)
20  ;INT(FNSEC(PI/4%))
RUN¶
 1
BASIC
```

Ex. 2

```
LIST¶
 5  EXTEND
10  ! THIS EXAMPLE COMPUTES THE
20  ! VOLUME OF SPHERE WITH RADIUS(R) IN
30  ! RANGE: 1<=R<=4.
40  OPEN "CON:" AS FILE 1
50  DEF FNSPVOL
60    FOR R=1 TO 4
70      X=R**3
80      Y=PI*X
90      Z=4/3
100      VOL=Y*Z
110      ; #1 FIX(VOL)
115   NEXT R
116   CLOSE
120   RETURN 0
130 FNEND
140 X=FNSPVOL
150 ; "END "
160 END
RUN¶
 4
 33
 113
 268
END
BASIC
```

PDL Function

Function: Returns a specific joystick's x or y coordinate or a value specifying which joystick control button was depressed.

Mode: Program/Direct

Format: PDL(<expression>)

Argument: <expression> is an integer expression which can have the following values:

0% - Reads x coordinate for joystick 1
1% - Reads y coordinate for joystick 1
2% - Reads x coordinate for joystick 2
3% - Reads y coordinate for joystick 2
4% - Reads which joystick and control button was depressed as follows:

| Button Depressed | Value Returned |
|---|---|
| 1 | 1 |
| 2 | 4 |
| 1 and 2 | 5 |

Result: Integer

Use: There are two possible joysticks: 1 and 2. Each joystick contains a control lever and a control button identified here as A and B. PDL allows the system to read and return the x and y coordinates of each joystick's control level (y = 0 to 239, x = 0 to 239. It also enables Monroe BASIC to return a value specifying which one or all of the control buttons were depressed.

Examples:

Ex. 1

```
;PDL(4%)
 5  (Assume all control buttons were depressed.)
```

Ex. 2

```
10  X%=PDL(0%)¶
20  Y%=PDL(1%)¶
30  ;"S=" X%,"Y= " Y% ! RETURNS HI-RES COORDINATES¶
40  END¶
RUN
 X= 60 (position 10)
 Y= 190 (line 5)
BASIC
```

Ex. 3

```
LIST¶
10 ! USE JOYSTICK TO DRAW FIGURE ON SCREEN
20 FGTCL 130 ! SELECT COLOR GROUP
30 FGPOINT 0,0,0
40 FGFILL 239,239 ! CLEAR SCREEN
50 FGLINE PDL (0),PDL(1),1 ! DRAW LINE
60 I%=I% + 1% ! INCREMENT COUNTER
70 IF I < 500 THEN GOTO 50
80 END
BASIC
```

REM Function

Function: Inserts comments into a user's program.

Format:
```
REM [remark]
    or
! [remark]
```

Argument: Remark can contain any printing characters on the keyboard. The Monroe BASIC interpreter completely ignores anything on a line following the letters REM or !.

Result: Must be used as a statement.

Use: It is often desirable to insert notes and messages within a user program. Documenting a program enables easy referencing by anyone using the program. REM statements do not offset program execution.

Example: Typical REM statements are shown below:

```
10   REM ...THIS PROGRAM CALCULATES MEAN VALUES..
20   ! ***MEAN VALUES ARE AVERAGE VALUES***
30   DEF FNSEC(X)=1/SIN(X)!DEFINE SECANT FUNCTION
```

Remarks are printed when the user program is listed.

## SLEEP Function

**Function:** Suspends the currently running program for a specified number of seconds. At the end of this period the program resumes execution.

**Mode:** Program.

**Format:** SLEEP <expression>

**Argument:** The value of the expression determines the number of seconds.

**Result:** None, must be used as a statement.

**Example:**

```
10 FOR I = 0 TO 100¶
20 NEXT I¶
30 ;I¶
40 T$="1981-6-2 10:10:00"
50 SET TIME T$
60 ;TIME$
70 SLEEP (10) !10 SECOND DELAY¶
80 PRINT "GOOD-BYE"¶
90 ;TIME$
100 END¶
RUN¶
 101        (Note:  10 second delay)
 1981-06-02 10:10:00
 GOOD-BYE
 1981-06-02 10:10:10
BASIC
```

SOUND Function

Function: Generates sounds on system speakers with specified qualities.

Mode: Program/direct.

Format: SOUND <channel%>,<pitch%>,<atten%>

Argument: Channel is the number of the sound or tone generator desired. It can be an integer constant or an integer variable. Values of 1,2 or 3 determine which tone generator is desired while 4 specifies the noise generator. All previous sound or tone generators can be turned off if a value of "0" is used and the rest of the parameters are omitted (e.g., SOUND 0%).

Pitch is a divide factor used to change the tune frequency (base to mid range to treble) of the sound produced. It is an integer variable. Values between 1 (treble) to 1024 (base) are acceptable. A value of "0" can also be specified but is normally only used on channel 4 (noise) to avoid clicks in "CRASH" sounds when the attenuation is changed.

Atten is the desired amount of attenuation of the sound that is produced. It can be an integer variable or constant. Values between 0 (no attenuation) to 15 (no sound) are acceptable and produce the following attenuations (DBs).

| Value | Attenuation (DB's) | Value | Attenuation (DB's) |
|---|---|---|---|
| 0 | 0 | 8 | 16 |
| 1 | 2 | 9 | 18 |
| 2 | 4 | 10 | 20 |
| 3 | 6 | 11 | 22 |
| 4 | 8 | 12 | 24 |
| 5 | 10 | 13 | 26 |
| 6 | 12 | 14 | 28 |
| 7 | 14 | 15 | sound off |

Result: Must be used as a statement.

Use: The SOUND function can be used to generate tones, clicks, buzzes, etc. of various pitches, loudness and duration. They can be combined to give still more variations. When SOUND function follows another, the resulting sound continues without interruption but mixes with the previous ones.

Example:

```
LIST¶
 5  INPUT "TIME" T%
10  INPUT "A FRE" A%
20  INPUT "B FRE" B%
30  INPUT "C FRE" C%
40  FOR J% = 1% TO T%
50    SOUND 1%, A%, 0%
60    SOUND 2%, B%, 0%
70    SOUND 3%, C%, 14%
80  NEXT J%
90  END
BASIC
```

TAB Function

Function: Tabulates to the specified position on a line.

Mode: Program/Direct.

Format: TAB(<expression>)

Argument: Expression is evaluated to an integer.

Note: TAB must be preceded on the program line by a ";" or "PRINT". TAB must be used in a PRINT statement.

Use: TAB can only be used with the PRINT statement. The first position on a line is position 1. The position specified by expression is always relative to position 1.

More than one TAB can appear on a line. In this case, TABS and items to be printed are normally separated by a semicolon or nothing at all. If expression evaluates to a position number to the left of the current position, that TAB will be executed at the specified position on the next line. If expression evaluates to a position greater than the screen width, expression will be treated as modulo the screen width.

Example:

```
;TAB(1)"HHHH"¶
 HHHH
BASIC
10  REM ***TAB SPACING***¶
20  F=2 : G=5¶
30  ; TAB(F)"X";TAB(G)"Y";TAB(G)"A";TAB(F)"B"¶
RUN¶
 X  Y
    A
 B
BASIC
```

## TIME$ Function

**Function:** Returns year-month-day and hour:minutes:seconds.

**Mode:** Program/direct.

**Format:** TIME$

**Use:** TIME$ is used in conjunction with the SET TIME$ statement. SET TIME$ sets the time and date and TIME$ reads it when desired in a program.

**Result:** String

**Example:**

```
LIST¶
10   INPUT "N=" N
20   DIM T$=25%
30   T$="1981-6-2 10:12:00"
40   SET TIME T$
50   ; TIME$
60   GET X$
70   FOR I=1 TO N
80   ; CHR$(7); ! BELL SOUNDS
90   NEXT I
100  ; TIME$
110  END
RUN¶
 N=100¶
 1981-06-02 10.12.00
 ¶
 .
 .
 .
 1981-06-02 10.12.06
BASIC
```

## CURREAD Function

**Function:** Reads the current cursor position.

**Mode:** Program/direct.

**Format:** CURREAD <yposition, xposition>

**Arguments:** yposition is a variable which will be set to the value (from 0 to 23) of the current line where the cursor is positioned.

xposition is a variable which will be set to the value (from 0 to 39/79, 39 is valid for the educational model and 79 for the business model) of the current position on the line where the cursor is positioned.

**Use:** This function reads the current cursor position and assigns its values to specified variables.

**Note:** The previous values of the yposition and xposition variables are lost once this function is given.

**Example:**

```
LIST¶
10  PRINT CHR$(12) ! CLEAR THE SCREEN
20  PRINT CUR(12,15) "T"; ! PRINT ON 13th LINE, 16th COLUMN
30  CURREAD Ypos,Xpos ! READ CURSOR POSITION
40  PRINT ! SKIP TO NEXT LINE
50  PRINT Ypos,Xpos ! THEN PRINT OLD CURSOR POSITION
60  END
BASIC
RUN¶
                T
 12              16
BASIC
```

# SECTION 11
# FORMATTED PRINTING

# SECTION 11
# FORMATTED PRINTING

## 11.1 INTRODUCTION

The PRINT USING statement can be employed in situations where a specific output format is desired. This situation might be encountered in such applications as printing payroll checks or accounting reports.

Format: PRINT USING [#<channel no.>,] <string1>;<value list>
or
; USING [#<channel no.>]<string1>;<value list>

The string may be a string variable, string expression, or a string constant, which is a precise image of the line to be printed. All the characters in the string are printed just as they appear, with the exception of the formatting characters. The value list is a list of the items to be printed. The string is repeatedly scanned until (1) the string ends and there are no values in the value list, or (2) a field is scanned in the string, but the value list is exhausted. If more than one format is included in the string, the first list item will use the first format, the second item the second format and so on. The string is constructed according to the rules listed in this section. The string may be separated from the value list by a comma, semicolon or nothing.

Note that the "OPTION EUROPE 1" statement can be specified before the "PRINT USING" statement when European notation is desired. Refer to Section 8 for details. The formatting characters for this option are specified in Section 11.3.

## 11.2 STRING FIELDS

When strings are to be printed via "PRINT USING", one of the following three formatting characters may be specified:

| Character | Function |
|---|---|
| "!" | Specifies that only the first character in the given string is to be printed. |

Example:

```
10  A$="LOOK"¶
20  PRINT USING "!";A$¶
RUN¶
L
BASIC
```

| | |
|---|---|
| \ n spaces \ | Specifies that the first 2+n characters from the string are to be printed. If the "\" characters are typed without any spaces, two characters will be printed and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left justified in the field and padded with spaces to the right. |

Example:

```
10  A$="LOOK" :B$="OUT"¶
20  PRINT USING "\\ ";B$¶
30  PRINT USING "\\ ␣";A$,B$¶  (Note:  ␣ = blank space)
40  PRINT USING "␣\␣␣\␣";A$,B$,"!!"¶
RUN¶
OU
LO OU
 LOOK OUT      !!
```

&

Specifies a variable length string field. When the field is specified with "&", the string is output exactly as is.

Example:

```
10  A$="LOOK": B$="OUT"¶
20  PRINT USING "!";A$¶
30  PRINT USING "&";B$¶
RUN¶
L
OUT
BASIC
10  A$="LOOK"¶
20  PRINT USING "& &," CUR (10,10)B$¶
30  ! B$ WILL BE PRINTED AT LINE 10, COL 10¶
```

## 11.3 NUMERIC FIELDS

The following formatting characters can be used to format a numeric field:

| Character | Use |
|---|---|
| # | A number character # is used to represent each digit position. All digit positions will be filled. If the number to be printed has fewer digits than the positions specified, the number will be right-justified (preceded by spaces) in the field. |

Example:

```
PRINT USING "####";88¶
  ␣␣88
```

| | |
|---|---|
| . | A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (0 if necessary). The numbers will be rounded off if necessary. Note that by including the "OPTION EUROPE 1" statement, the decimal point in a numeric field will be replaced by a comma. |

Examples:

```
PRINT USING "##.##";.08¶
0.08

PRINT USING "###.##";887.654¶
887.65

PRINT USING "##.## ";20.2,7.3,88.789,.567¶
20.20      7.30  88.79   0.57

DOUBLE¶
OPTION EUROPE 1¶
PRINT USING "#######.##"; 1.23456
1234567,89
```

| Character | Use |
|---|---|

\+ The "+" sign may be used at either the left or the right of the numeric field. If the number is positive, the + sign is printed at the specified side of the number. If the number is negative, a - sign is printed at the specified side of the number.

Example:

```
PRINT USING "+##.##";-75.95,2.5,88.6,-.8¶
-75.95  +2.50+88.60  -0.80
```

\- The "-" sign, when used at the right of the numeric field, prints to the right of a negative number. If the number is positive, a space is printed.

Example:

```
PRINT USING "##.##- ";-75.95,44.449,-8.01¶
75.95- 44.45  8.01-
```

** The "**" placed at the beginning of a numeric field fills the unused spaces in the leading portion with asterisks. The "**" also specifies positions for two more digits (termed "asterisk fill").

Example:

```
PRINT USING "**#.#  ";22.39,-0.8,543.1¶
*22.4  *-0.8  543.1
```

$$ When the $$ is used at the beginning of a numeric field a $ sign is printed in the space immediately preceding the number printed. Note that $$ also specifies positions for two more digits, but that the $ sign itself takes up one of these spaces.

Example:

```
PRINT USING "$$###.## ";123.45¶
$123.45
```

| Character | Use |
|---|---|
| **$ | The combination "**$" at the beginning of a format string combines the effects of ** and $$. Leading spaces will be filled with asterisks and a dollar character will be printed before the number. "**$" specify three more digit positions, one of which is the dollar character. |

Example:
```
PRINT USING "**$##.##";2.34¶
***$2.34
```

| | |
|---|---|
| , | A comma to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma at the end of the format string is printed as part of the string. This comma serves as the delimiter between two numbers. A comma specifies one digit position. Note that by including the statement, "OPTION EUROPE 1", the decimal point in a numeric field will be replaced by a comma. |

Examples:
```
PRINT USING "#,###.##";4567.8¶
4,567.80

PRINT USING "####.##";4567.8¶
4567.80

DOUBLE¶
OPTION EUROPE 1
PRINT USING "#,###,###.##";1.23456789E+06¶
1.234.567,89
```

| | |
|---|---|
| ^^^^ | Four carats may be placed after the digit position characters to specify exponential format. The four carats specify the position of E+xx. Any decimal point position may be specified; the exponent will be adjusted. Unless a leading + or leading or trailing + or - are specified, one digit position at the beginning of the number will be used to print the minus sign. |

Examples:

```
PRINT USING "##.##^^^^";123.45¶
1.23E+02
PRINT USING ".####^^^^";-777777¶
%-777777
PRINT USING "+.##^^^^";234
+.23E+03
```

\- An underscore in the format string causes the next character to be output as a literal character. The literal character itself may be an underscore. The underscore is used to print the characters which would otherwise be interpreted as format characters.

Example:

```
PRINT USING "_!##.##_!";45.67¶
!45.67!
```

Note that if the number to be printed is larger than the specified numeric field, a percent character is printed before the number. A percent character is printed also if rounding causes a number to exceed the field.

Examples:

```
PRINT USING "##.##";711.22¶
% 711.22

PRINT USING ".##";.999¶
% .999
```

% A "%" sign may replace the "," in a numeric field format in order to insert a blank where the "," (or "." with "OPTION EUROPE 1") was to be in the output. This can be used, for example, for special forms.

Examples:

```
DOUBLE¶
PRINT USING "#%###%###";1.234567E+06¶
1 234 567
```

~ A "~" may replace the "." in a numeric field format to insert a blank where the "." (or "," with "OPTION EUROPE 1") was to be in the output. This can be used, for example, for special forms.

Examples:

```
DOUBLE¶
PRINT USING "#%###%###~###";1.23456789E+06¶
1 234 567 890
```

## 11.4 ILLUSTRATED EXAMPLE

The following program illustrates the formatting rules presented in this section.

```
10 INPUT A$,A¶
20 PRINT USING A$;A¶
30 GOTO 10¶
RUN¶
```

(The screen displays a "?". The numeric field and value list are entered and the output is displayed.)

```
? +#,9¶
+9
? +#, 10¶
% 10
? ##,-2¶
-2
? +#,-2¶
-2
? +#.##,9
+9 -2
? #,-2¶
%-2
? +.###,.02¶
+.020
? ####.#,100¶
 100.0
? ##+,2¶
 2+
? THIS IS A NUMBER ##,2¶
THIS IS A NUMBER 2
? BEFORE ## AFTER,12¶
BEFORE 12 AFTER
? ####,44444¶
% 44444
```

```
? **###,1¶
***1
? **###,12¶
**12
? **###,123¶
*123
? **###,1234¶
1234
? **###,12345¶
% 12345
? **,1¶
*1
? **,22¶
22
? **.##,12¶
12.00
? **#####,1¶
*****1
? $####.##,12.34¶                    (Note: not floating $)
$  12.34
? $$####.##,12.56¶                   (Note: floating $)
   $12.56
? $$.##,1.23¶
$1.23
? $$.##,12.34¶
% 12.34
? $$###,0.23¶
   $0
? $$####.##,0¶
   $0.00
? **$###.##,1.23¶
****$1.23
? **$.##,1.23¶
*$1.23
? **$###,1¶
****$1
```

```
? #,6.9¶
7
? #.#,6.99¶
7.0
? ##-,2¶
 2
? ##-,-2¶
 2-
? ##+,2¶
 2+
? ##+,-2¶
 2-
?##^^^^,2¶
 2E+00
?"##^^^^",12¶
 1E+01
?"#####.###^^^^",2.45678¶
 2456.780E-03
?"#.###^^^^",123¶
 0.123E+03
?"#.##^^^^",-123¶
 -.12E+.03
? "#####,###.#",1234567.89¶
1,234,567.9
.
.
.
```

(Depressing the STOP key stops the program.)

# SECTION 12
# LOW RESOLUTION COLOR GRAPHICS

# SECTION 12
# LOW RESOLUTION COLOR GRAPHICS

## 12.1 COLOR GRAPHICS KEYWORDS

Low resolution color graphics is available on the Monroe 8800 Series Educational Computer. The selection of display colors for low resolution text or graphics is enabled by including special control keywords in the PRINT statement. The PRINT statement affects one display line at a time. A summary of these keywords is shown in Table 12-1.

Table 12-1. Low Resolution Color Graphics Keywords

| Keyword | Function |
|---|---|
| RED (Red) MAG (Magenta)<br>GRN (Green) CYA (Cyan)<br>YEL (Yellow) WHT (White)<br>BLU (Blue) | Displays background or alphanumeric variables, expressions or text strings in specified color. |
| GRED GMAG<br>GGRN GCYA<br>GYEL GWHT<br>GBLU | Displays graphics via text strings in specified color. |
| FLSH | Displays text or graphics in flashing mode. |
| STDY | Negates flashing mode. |
| DBLE | Displays text or graphics in double height mode. |
| NRML | Negates double height mode. |
| NWBG | Enables background color to be changed. |

Table 12-1. Low Resolution Color Graphics Keywords (Cont.)

| Keyword | Function |
|---|---|
| BLBG | Changes background color to black. |
| GCON | Displays graphics characters in contiguous mode. |
| GSEP | Displays graphic characters in separated mode. |
| GHOL | Enables control blanks on screen to be filled with previous graphic characters. |
| GREL | Negates previous GHOL keyword. |
| HIDE | Allows graphics to be hidden in display. |

Each of the control keywords in Table 12-1 places a control character on the screen. Although these characters are not visible, they take up one position each on the current line. Specified control characters can be covered by a background color if the control keywords are given in the correct order. Detailed descriptions including correct keyword order can be found in subsequent paragraphs.

The available low resolution graphics characters in the system are listed in Appendix E. This table gives the ASCII value of each character and its meaning in character mode and in graphic mode.

Note that the capital letters still remain the same in graphic mode. You can mix capital letters and graphic characters just as you like.

Keyword: <color>

Function: Displays alphanumeric characters in designated color.

Mode: Direct/Program.

Format: PRINT [position] <color> <list>

Arguments: Position can be of the form CUR(y,x) or TAB(con), where:
y,x is the row and column where the cursor is to be moved and color alphanumeric characters are to be displayed (0,0 is the top left corner of screen).

- Con is the stated horizontal position on the current row where color alphanumeric characters are to be displayed. The first position of a line is position 1. An expression may be substituted for con.

List can contain variables, expressions, or text strings.

Color can be any one of the following seven colors:

RED = red
GRN = green
YEL = yellow
BLU = blue
MAG = magenta
CYA = cyan
WHT = white

Use:

This command causes alphanumeric text of the user's choice to be displayed in color. It can be given and reissued at any point in a program; however, the character position (x,y) at the current line where a PRINT... color... command is given is lost for display purposes (refer to example below).

Examples:

Ex. 1

```
10  ; CHR$ (12%)¶
20  ; "THIS LINE PRINTS IN WHITE (DEFAULT)."¶
30  ; RED "THESE TWO LINES PRINT IN RED AND"¶
40  ; RED "ONE CHARACTER TO THE RIGHT."¶
50  END¶
RUN¶
BASIC

THIS LINE PRINTS IN WHITE (DEFAULT).
  THESE TWO LINES PRINT IN RED AND
  ONE CHARACTER TO THE LEFT.
```

Ex.2 The statement:

```
PRINT CUR (5,12) GRN "CHART OF ACCOUNTS"
```

will display "CHART OF ACCOUNTS" in green beginning in row 5 column 12.

Ex. 3 The statement:

```
PRINT TAB (7) YEL "ADDRESS:"
```

will display "ADDRESS:" in yellow in pos 7.

Keyword: NWBG

Function: Selects new background color of display.

Mode: Direct/Program.

Format: PRINT [position] <color> NWBG <list>

Arguments: Position can be of the form CUR(y,x) or TAB(con), where:
y,x is the row and column where the cursor is to be moved and color alphanumeric characters are to be displayed (0,0 is the top left corner of screen).

Con is the stated horizontal position on the current row where color alphanumeric or graphic characters are to be displayed. The first position on a line is position 1. An expression may be substituted for con.

Color can be any one of the following seven colors:

| | | | | | |
|---|---|---|---|---|---|
| RED | = | red | MAG | = | magneta |
| GRN | = | green | CYA | = | cyan |
| YEL | = | yellow | WHT | = | white |
| BLU | = | blue | | | |

List can contain variables, expressions, or text strings.

Use: The background of a display is normally black. A background of any of the standard colors can be selected using the NWBG keyword. After this keyword is given, the background becomes the color of the preceding color keyword. This facility allows letters, numbers and graphics to be highlighted and shaded. It can also be used for prompting purposes. A black background can be restored by issuing the BLBG keyword.

Note that the color of the background should be specified first followed by the NWBG keyword. If this order is adhered to the new background color will start one character to the right of where it is expected on the current line. That is, the attribute NWBG occupies one print position. If TAB or CUR is included that print position will still be reserved for the keyword and should not be used.

Examples:

Ex. 1

The following statements use a yellow prompt (NAME?) on a blue background. "JOHN SMITH" prints in red on a black background.

```
10  ; CHR$ (12%)
20  ; BLU NWBG YEL "NAME?ÞÞ"
30  ; BLBG RED "JOHN SMITH"
```

Ex. 2

```
; "THIS PRINTS WHITE LETTERS ON A BLACK
 BACKGROUND"
```

Ex. 3

The following statements set the background color for the entire screen to cyan.

```
10  FOR I=0 to 23
20  ; CYA NWBG
30  NEXT I
```

Keyword: <gcolor>

Function: Displays graphic characters in designated color.

Mode: Program/Direct.

Format: PRINT [position] <gcolor> <glist>

Arguments: Position can be of the form CUR(y,x) or TAB(con), where:
y,x is the row and column where the cursor is to be moved and color graphic characters are to be displayed. The first position on a line is position 1. An expression may be substituted for con.

Con is the stated horizontal position on the current row where color graphic characters are to be displayed. The first position on a line is position 1. An expression may be substituted for con.

List can contain variables, expressions, or text strings.

Gcolor can be any one of the following seven colors:

| | |
|---|---|
| red = GRED | magenta = GMAG |
| green = GGRN | cyan = GCYA |
| blue = GBLU | white = GWHT |
| yellow = GYEL | |

glist is a text string which contains one or more of the 64 graphic characters that are available (refer to Appendix E).

Use: This keyword permits the construction of extra large letters and graphic diagrams. Note that the graphic character is displayed one additional character to the right of where it is expected to be. That is, the attribute <gcolor> occupies one print position. This reserved space can be eliminated via the GHOL keyword.

Examples: Ex. 1 The following statements print the block letter "L" in magenta.

```
10  ; CHR$ (12%)
20  ; TAB (15) GMAG "5"
30  ; TAB (15) GMAG "5"
40  ; TAB (15) GMAG "up"
50  END
```

Ex. 2 The statement:

```
; GRED "%" GHOL GGRN "%"
```

prints the graphic character "%" in red three times then once in GREEN.

Keyword: FLSH and STDY

Function: Displays alphanumeric or graphic characters in flashing color.

Mode: Direct/Program.

Format: PRINT [position] FLSH <color>

Arguments: Position can be of the form CUR(y,x) or TAB(con), where:
y,x is the row and column where the cursor is to be moved and color characters are to be displayed (0,0 is the top left corner of screen).

Con is the stated horizontal position on the current row where color alphanumeric characters are to be displayed. An expression may be substituted for con.

List can contain variables, expressions, or text strings.

Colors can be any of the following alphanumeric or graphic colors:

| Color | Alphanumeric | Graphic |
|---|---|---|
| Red | RED | GRED |
| Green | GRN | GGRN |
| Blue | BLU | GBLU |
| Yellow | YEL | GYEL |
| Magenta | MAG | GMAG |
| Cyan | CYA | GCYA |
| White | WHT | GWHT |

Use:

The keyword, FLSH, causes alphanumeric or graphic text to be displayed in a flashing mode. Note that the graphic character is displayed one additional character to the right of where it is expected to be. That is, the attribute FLSH or STDY occupies one print position. This reserved space can be eliminated via the GHOL keyword.

The STDY keyword disables the FLSH mode.

Example:

The statement:

```
; CUR(0,4) RED FLSH DBLE "MONROE" STDY
  "SYSTEMS FOR BUSINESS"
```

prints "MONROE" in a red, flashing and double height mode. "SYSTEM FOR BUSINESS" does not flash. STDY disables the flashing mode.

Keyword: DBLE and NRML

Function: Display alphanumeric or graphic characters in designated color with twice the normal height but with the same width.

Mode: Program/Direct.

Format: PRINT [position] DBLE <color> <list>

Arguments: Position can be of the form CUR(y,x) or TAB(con), where:
y,x is the row and column where the cursor is to be moved and color characters are to be displayed (0,0 is the top left corner of screen).

Con is the stated horizontal position on the current row where color alphanumeric or graphic characters are to be displayed. The first position on a line is position 1. An expression may be substituted for con.

List can contain variables, expressions, or text strings.

Colors can be any of the following:

| Color | Alphanumeric | Graphic |
|---|---|---|
| Red | RED | GRED |
| Green | GRN | GGRN |
| Blue | BLU | GBLU |
| Yellow | YEL | GYEL |
| Magenta | MAG | GMAG |
| Cyan | CYA | GCYA |
| White | WHT | GWHT |

Use:

The double-height alphanumeric and graphic characters displayed by this keyword can be used to highlight a particular section on the screen and for prompts. Note that the character width remains the same. This keyword is in effect for only the lines (rows) where it is given and also occupies one print position. When DBLE is in use, it causes the information for the row below it to be ignored and blanked out.

The keyword NRML when included causes the characters to be displayed in normal height.

Examples:

Ex. 1

```
; CUR (3,3) DBLE "THIS LINE PRINTS IN DOUBLE
  HEIGHT."
```

Ex. 2

```
; CUR (0,0) YEL DBLE "MONROE" NRML "SYSTEMS FOR
  BUSINESS"
```

Displays "MONROE" in yellow in double height on a black background.

Keyword: GSEP and GCON

Function: Display graphic characters in separate (dotted line) or contiguous configuration.

Mode: Direct/Program.

Format: PRINT [position] <gcolor> GSEP <list>

Arguments: Position can be of the form CUR(y,x) or TAB(con), where:
y,x is the row and column where the cursor is to be moved and color graphic characters are to be displayed (0,0 is the top left corner of screen).

Con is the stated horizontal position on the current row where graphic characters are to be displayed. The first position on a line is position 1. An expression may be substituted for con.

Gcolor can be any one of the following seven colors:

| | | | | | |
|---|---|---|---|---|---|
| red | = | GRED | magenta | = | GMAG |
| green | = | GGRN | cyan | = | GCYA |
| blue | = | GBLU | white | = | GWHT |
| yellow | = | GYEL | | | |

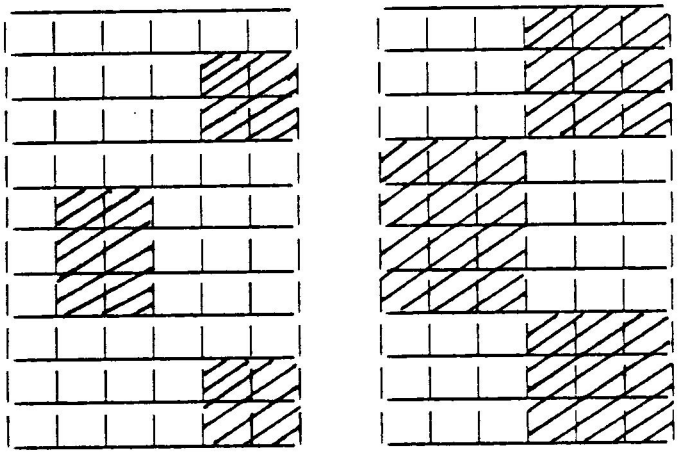
List can contain expressions or text strings.

Use: The GSEP keyword allows specific areas of graphic characters to be distinguished by being separated (dotted).

Figure 12-1 shows the difference between separate and normal (contiguous) configuration, for graphic character "f". The graphic display will start at least two characters to the right (one for <gcolor> and one for GSEP keywords) of where it is expected on the current line. That is, the attributes gcolor

and GSEP occupy two print positions. If TAB or CUR is included the current two character positions on the specified line will be reserved and shall not be used.

The use of the keyword GCON returns the character to to the contiguous configuration.



(a) Separate (b) Contiguous

Figure 12-1. Graphics Character Generation

Example:

Ex. 1

```
; GCYA GSEP "f"
```

will produce Figure 12-1 (a).

Ex. 2

```
; GCYA GCON "f"
```

will produce Figure 12-1 (b).

Keyword: GHOL and GREL

Function: Fills the blanks on the graphic display produced by the previous graphic keywords with a repeat of the preceding graphic character.

Mode: Program/Direct.

Format: PRINT [position] <gcolor> <list> GHOL

Arguments: Position can be of the form CUR(y,x) or TAB(con), where:
y,x is the row and column where the cursor is to be moved and color graphic characters are to be displayed (0,0 is the top left corner of screen).

Con is the stated horizontal position on the current row where color alphanumeric characters are to be displayed. The first position on a line is position 1. An expression may be substituted for con.

Gcolor can be any one of the following seven colors:

| | | | |
|---|---|---|---|
| red | = GRED | magenta | = GMAG |
| green | = GGRN | cyan | = GCYA |
| blue | = GBLU | white | = GWHT |
| yellow | = GYEL | | |

List can contain variables, expressions, or text strings.

Use: This keyword allows different colored areas in graphics and diagrams to be connected without gaps in between colors. Without this keyword gaps would appear between colors, background changes, etc.

The keyword GREL (release graphic) cancels the GHOL keyword. This keyword is in effect for only the line where it is given.

Example: ; GREL "%" GHOL

which is the same as

; GREL "%%"

Keyword: HIDE

Function: Enables graphics to be hidden by displaying graphics in the same color as the background.

Mode: Program/Direct.

Format: PRINT [position] <gcolor> HIDE <glist>

Arguments: Position can be of the form CUR(y,x) or TAB(con), where:
y,x is the row and column where the cursor is to be moved and color alphanumeric characters are to be displayed (0,0 is the top left corner of screen).

Con is the stated horizontal position on the current row where color alphanumeric characters are to be displayed. The first position on a line is position 1. An expression may be substituted for con.

Gcolor can be any one of the following seven colors:

| | | | | | |
|---|---|---|---|---|---|
| red | = | GRED | magenta | = | GMAG |
| green | = | GGRN | cyan | = | GCYA |
| blue | = | GBLU | white | = | GWHT |
| yellow | = | GYEL | | | |

Glist can contain variables, expressions, or text strings.

Use: This keyword can be used to animate graphic displays or to hide answers to questions until the user responds accordingly. The hidden graphic will start at least two characters to the right (one for <gcolor> and one for HIDE keywords) of where it is expected on the current line. That is, the attributes gcolor and HIDE occupy two print positions. If TAB or CUR is included, the current

two-character positions on the specified line will be reserved and should not be used. Refer to GHOL keyword to see how these reserved spaces can be filled.

Example:

```
LIST¶
10  ; CHR$ (12%)
20  A$(1) = "THESE" : A$(2) = "WORDS"
30  A$(3) = "WERE" : A$(4) = "HIDDEN"
40  FOR I=0 TO 23 : REM DISPLAY CYAN BACKGROUND
50  ; CYA NWBG
60  NEXT I
70  C$ = BLU : H$=HIDE : X=0
80  Z=FNA : REM HIDE THE ANSWERS
90  ;CUR(0,0)
95  INPUT "TYPE Y TO CONTINUE ";Y$
.
.
.
500 H$=FLSH : X=0
510 Z=FNA : REM FLASH THE ANSWERS
520 DEF FNA
530 FOR I=4 TO 19 STEP 5
540 X=X + 1
550 ; CUR (I,I) C$ H$ A$(X)
560 NEXT I
570 RETURN 0
580 FNEND
.
.
.
999 STOP
```

## 12.2 COLOR GRAPHICS STATEMENT AND FUNCTION

The following statement and function is available to write, delete, or test for the presence of a graphic point on the screen:

| | |
|---|---|
| TXPOINT | Writes, deletes or tests for a point on the screen. |

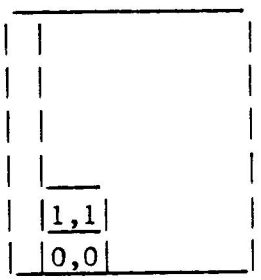
## TXPOINT Function

**Function:** Returns a value specifying whether a particular graphic point was turned on or off.

**Mode:** Program/Direct.

**Format:** TXPOINT (<xvar>,<yvar>)

**Argument:** xvar, yvar is the location on the screen which is to be tested. These variables can take on values from 0 to 78 for x and 0 to 71 for y, where the point (0,0) is at the bottom left, one column to the right as shown below.




**Use:** An expression may be substituted for the variable. This function returns a "-1" if the point is set and a "0" is not set.

**Example:**

```
10  Z% = TXPOINT (6,10)¶
20  IF Z% <> 0 THEN TXPOINT 6,10,-1¶
```

If the TXPOINT is set at (6,10), the graphic point is turned off.

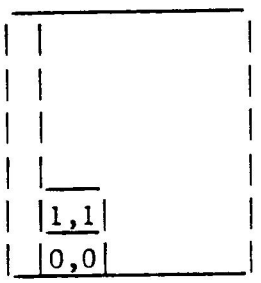
## TXPOINT Statement

**Function:** Sets (writes) or deletes a specific graphic point on the screen.

**Mode:** Program/Direct.

**Format:** TXPOINT <xvar>,<yvar>[,-1]

**Argument:** xvar, yvar is the x,y coordinate on the screen which is to be set or deleted. This variable can take on values from 0 to 78 for x and 0 to 71 for y, where the point (0,0) is at the bottom left, one column to the right as shown below.




-1, when included, deletes instead of writes the point.

**Use:** This statement can be used to turn on or turn off any one or all of the 71 by 78 blocks on a screen.

**Example:** Ex.1 The following statements display a red point at (0,0). That is, at the second print position on line 0.

```
LIST¶
10  ; CHR$ (12%) : REM CLEAR LOW RES SCREEN
20  ; CUR (23, 0) GRED;
30  TXPOINT 0,0
BASIC
```

Ex. 2 The following program causes a SINE curve to be displayed with all seven colors using the TXPOINT statement.

```
LIST¶
10  INTEGER
20  DIM A$(7%) ! DEFINE 7 COLORS FOR LO-RES DISPLAY
30  A$(0%)=GRED
40  A$(1%)=GGRN : A$(2%)=GYEL : A$(3%)=GBLU
50  A$(4%)=GMAG : A$(5%)=GCYA : A$(6%)=GWHT
60  INPUT "m,r ="M%,R
63  ! r is used to set # of periods (select a value from
64  ! .3 to 2)
65  ! m = # of frequency increments (select a value
66  ! from 1 to 8)
70  S=2%*PI/(R*77%) ! INCREMENT FOR ARGUMENT OF SINE
80  PRINT CHR$(12%) ! CLEARS SCREEN
90  FOR I%=0% TO 23% ! SELECT COLOR FOR EACH LINE
100     PRINT CUR(I%,0%) A$(MOD(I%,7%));
110 NEXT I%
120 FOR I%=2% To 77% ! FOR EACH COLUMN COMPUTE
125 ! COORDINATES AND DISPLAY POINTS OF SINE CURVE
130     GOSUB 170
140 NEXT I%
150 PRINT CUR(0%,15%) RED FLSH DBLE "sine"
160 END
170 FOR J%=0% TO M%
180     TXPOINT I%,32%+SIN((I%+J%)*S)*30%
190 NEXT J%
200 RETURN
210 END
BASIC
```

## 12.3 STRING MANIPULATION OF LOW RESOLUTION GRAPHICS

The following low resolution graphics functions actually generate strings which can be manipulated and be printed as strings:

| | |
|---|---|
| Alphanumeric colors: | RED, GRN, YEL, BLU, MAG, CYA, WHT |
| Graphic colors: | GRED, GGRN, GYEL, GBLU, GMAG, GCYA, GWHT |
| Graphic functions: | FLSH, STDY, DBLE, NRML, MWBG, BLBG, GCON, GSEP, GHOL, GREL, HIDE |

In addition, the CUR(row,column) function also returns a string value; however, the TAB function can only be used with a PRINT statement.

Of the functions listed above NWBG and HIDE use the previous character as either the new background or hidden color respectively.

Example:

```
A$=CUR(22,10)+ FLSH + RED + BLBG + 'ATTENTION'¶
; A$  ! Print a red flashing 'ATTENTION' on line
  22, column 10¶
```

# SECTION 13
# HIGH RESOLUTION
# COLOR GRAPHICS

# SECTION 13
# HIGH RESOLUTION
# COLOR GRAPHICS

## 13.1 INTRODUCTION

High resolution color graphics is available on the Monroe 8800 Series Educational Computer. For high resolution, the screen is divided into 240 x 240 individually accessible points called pixels. The origin of the screen (0,0) is in the lower left corner and each coordinate has a range of from 0 to 239.

Two data bits correspond to each pixel. When these bits are used to select one of four colors of a particular group, it is referred to as the four-color mode. High resolution graphics can be shown together with the usual text or graphics display. When the two data bits are considered to belong to two separate pictures each having only two colors, it is referred to as the animation mode. The animation mode is used to enable one picture to be generated while another is being shown. It is also used when very fast switching between two pictures is desired.

Table 13-1 lists the available high resolution graphic statements.

Table 13-1. High Resolution Graphics Statements

| Statement | Use |
|---|---|
| FGCIRCLE | Draws a circle. |
| FGGET | Copies a rectangle. |
| FGCTL | Selects color combination to be used. |
| FGDRAW | Displays a specified shape. |
| FGERASE | Sets all elements of a shape to its background color. |

| Statement | Use |
|---|---|
| FGFILL | Fills a rectangle with desired color. |
| FGLINE | Draws a straight line between two pixels. |
| FGPAINT | Fills a closed area with desired color. |
| FGPOINT | Sets or returns the color number of a specified pixel. |
| FGPUT | Restores a rectangle to high resolution memory. |
| FGROT* | Rotates specified shape in 45° increments. |
| FGSCALE* | Scales shape to be drawn. |

Once the four-color combination is specified by the FGCTL statement, other high resolution statements may specify a particular color in that group. Each of these statements contains a <color> argument. This variable can take on values between 0 and 3 and corresponds to the colors in that particular color group (see Table F-1). For example, consider color group 3 in Table F-1:

0 1 2 3
black, red, green, blue.

Here, 0 = black, 1 = red, 2 = green, and 3 = blue.

Color groups 0 to 127 are for both graphics text and high resolution color in a display and color groups 128 to 256 are for high resolution graphics only. Once a high resolution color group is specified (e.g., FGCTL 130) and the execution of the program has completed, a LIST command must be entered to get back to BASIC!

* Note that the FGROT and FGSCALE statements previously executed remain in effect until reset by a "FGROT 0" and "FGSCALE 1,1." Hence, it is a good practice to specify these reset statements at the beginning of each high resolution program.

## 13.2 ANIMATION MODE

Two colors are used in the animation mode. The following procedure can be used:

1. Pick out a color selection group (72-127). The color selection groups are used in pairs (e.g., 72-73, 74-75).

2. Draw a picture with color number 1 or 2. Select the same color as the one the picture is drawn on. The picture cannot be seen.

3. Change the color selection group so that the picture that was drawn in color number 1 or 2 becomes visible.

4. Draw a new picture according to step 2 above.

5. Change the color selection group so that the picture that was drawn in step 2 disappears and the one drawn in step 4 will show.

6. Erase the picture drawn in step 2 and draw a new one.

7. Change the color selection group so that the picture drawn in step 6 becomes visible.

Repeat the procedures under step 6 and 7.

To protect the current picture until a new picture is to be shown use the following method:

```
100 FGLINE 100,100,256*2+1
```

This instruction will cause a line to be drawn from the previous position to the point 100,100 with color number 1. Color number 2 is protected and will not be changed. Additional examples of this feature are shown under the FGLINE statement.

## 13.3 FGCIRCLE STATEMENT

**Function:** Draws a circle or a specified section.

**Mode:** Program/Direct.

**Format:** FGCIRCLE <x,y>[,length]

**Arguments:** x,y specifies the coordinates of the center of the circle or arc to be drawn. The radius is equal to the distance between x,y and the previous x,y.

Length equals the length of the arc to be drawn, starting at the line joining the center of the circle with the previous x,y and proceeding clockwise. If length is omitted, the full circle will be drawn.

If a negative length is given, the drawing of the arc will proceed counterclockwise instead of clockwise.

The length of the arc is expressed as a fraction of the circumference of the circle. The circumference is equal to 2*PI*R where:

$$R = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}$$

The coordinates $X_1$ , $Y_1$ are the previous x,y location (from FGPOINT) while $X_2$ , $Y_2$ are those specified by this statement.

**Use:** FGCIRCLE draws a circle or a specified section depending on whether the length parameter was included. Specifying a length greater then 2*PI*R has the same effect as if the length was omitted. After execution of the FGCIRCLE statement, the final point drawn will be the effective x,y.

Example:

```
LIST¶
 5 ;CHR$(12) ! CLEAR LO-RES SCREEN
10 FGCTL 2
20 FGPOINT 0,0,0
30 FGFILL 239,239
35 S%=1%
36 ! DRAW CIRCLES WITH ORIGIN MOVING FROM
37 ! LOWER LEFT CORNER TO THE CENTER.
38 ! THE POINT ON ARC FOLLOWS.
40 FOR I=110 TO 0 STEP -2
50   FGPOINT 110-I,10,1
55   ! THE LENGTH OF ARC IS A FULL CIRCLE.
60   FGCIRCLE 140-I,110-I,(120-I)*2*PI*S%
65   S%=S%*(-1)
70 NEXT I
BASIC
```

## 13.4 FGGET STATEMENT

**Function:** Copies the high resolution memory contents for the rectangle defined by the previous x,y coordinate and the x,y in FGGET.

**Mode:** Program/Direct

**Format:** <string var> = FGGET(x,y)

**Arguments:** x,y are the coordinates of the pixel directly opposite the previous x,y which together form opposite corners in a rectangle.

String var will contain header and graphics data for the rectangle to be copied. See "Use" below for an explanation of this data.

**Use:** The high resolution graphics data is described in a string by an 8-byte header and succeeding graphics data. It is used by FGPUT in restoring the rectangle stored by FGGET. This header can also be manipulated by the user to alter the restored rectangle's shape and relative position (refer to examples under FGPUT). The header's contents look like this:

| Byte | A$ |
|---|---|
| 1 | xpos |
| 2 | ypos |
| 3 | xsize |
| 4 | ysize |
| 5 | xst |
| 6 | yst |
| 7 | xcant |
| 8 | ycant |
| | Graphics data |

| Byte | Contents |
|---|---|
| 1-2 | xpos,ypos- The rectangle collected by FGGET is specified by the old position and the position in FGGET statement. The FGPUT restores the rectangle on the screen using the previous point. The relative positioning of the rectangle is maintained in FGPUT using xpos and ypos. They contain the displacements from old position to the upper left hand corner, where retrieval and restoration starts. xpos and ypos can be modified by the user to alter the relative positioning of the restored rectangle. |
| 3 - 4 | xsize,ysize- xsize describes the horizontal dimension in bytes. ysize is the number of lines in the defined rectangle. These are used in block transfers and memory allocation in FGPUT. They cannot be changed by the user. |
| 5 - 6 | xst,yst- xst indicates the position of upper left hand corner of the rectangle in a byte. Although a pixel in a non-byte boundry specified, the byte containing the pixel is collected. The xst is used to align the first pixel in FGPUT. yst is always assigned the value 0. |
| 7 - 8 | xcant,ycant- This data specifies the pixel counts in both x and y directions. These can be modified to change the shape of the rectangle restored by FGPUT. |
| 9 - ... | Graphics data. |

Example:

Ex. 1

```
 5 ;CHR$(12) ! CLEAR LO-RES SCREEN
10 FGCTL 2
20 FGPOINT 0,0,0
30 FGFILL 239,239
40 FGPOINT 10,10,1
50 FGFILL 90,90
60 FGPOINT 11,11,2
70 FGFILL 89,89
80 ! COLLECT THE RECTANGLE IN A$
90 FGPOINT 10,10,1
100 A$=FGGET(90,90)
110 ! PRINT CONTENTS OF HEADER
120 FOR I = 1 TO 8
120  N$ .= MID$(A$,I,1)
140  ; ASCII (N$)
150 NEXT I
160 END
RUN¶
0  (xpos)
80 (xpos)
21 (xsize)
81 (ysize)
2  (xst)
0  (yst)
81 (xcant)
81 (ycant)
BASIC
```

Example: Ex. 2

```
EXTEND¶
BASIC
LIST¶
 5 ;CHR$(12) ! CLEAR LOW-RES SCREEN
10 ! THIS PROGRAM DISPLAYS GROUPS OF RECTANGLES (10
20 ! x 10) IN RANDOM COLORS IN RANDOM POSITIONS ON
25 ! SCREEN.
30   DIM A$ (2%)
40   FOR II% = 130% TO 200%
50     FGCTL II%
60     FOR I% = 1% TO 3%
70       FGPOINT 0,0,I%
80       FGFILL 10,10
90       FGPOINT 0,0,I%
100      A$(I%-1%) = FGGET(10,10)
110  NEXT I%
120  FGPOINT 0,0,0
130  FGFILL 239,239
140  RANDOMIZE
150    FOR I%=1% TO 90%
160      X% = RND * 239%
170      Y% = RND * 239%
180      FGPOINT X%,Y%
190      J%=RND*2%
200      FGPUT A$(J%)
210    NEXT I%
220  NEXT II%
230  END
BASIC
```

NOTE: List command must be entered after execution of this program to get back to BASIC.

## 13.5 FGCTL STATEMENT

**Function:** Selects the mode (animation or four-color) and color combination to be used.

**Mode:** Program/Direct

**Format:** FGCTL <code>

**Arguments:** Code can take on values between 0 and 255 and represents the mode and color combination to be used. Refer to Appendix F for color selection table.

**Note:** Once a color group from 128 to 156 is specified and the execution of the program has completed, a LIST command must be entered to get back to BASIC!

**Use:** This statement allows the user to select the animation or four-color mode.

The <code> variable (see Table F-1) is in the interval 0 to 255. Values less than 128 mean that the ordinary text and graphics are displayed on top of the high resolution graphics. That is, at the point of intersection the low resolution and high resolution colors will mix. From 128 upwards the high resolution graphics memory is displayed. Color codes from 72 to 127 and 200 to 255 are used in the animation mode.

**Example:**

```
LIST  100-1100¶
100   FGCTL 0:REM SET COLORS TO ALL BLACK
120   FGPOINT 0,0,0 : REM SETS POINT (0,0)
130   FGFILL 239,239 : REM CLEAR SCREEN
140   FGCTL 3 : REM SELECTS COLORS - BK, R, GR & BL
.
.
.
1100  FGCTL 131 : REM HIGH RES DISPLAY FOR COLOR
      CODE 3 (128 + 3)
BASIC
```

## 13.6 FGDRAW STATEMENT

**Function:** Displays the shape at the previous x,y coordinates (subject to scaling and rotation) described by a table.

**Mode:** Program/Direct

**Format:** FGDRAW <string var>

**Arguments:** String var contains a shape table.

**Use:** Monroe BASIC has three statements - FGDRAW, FGSCALE and FGROT - which allow the user to manipulate shapes in high-resolution graphics. Before using these statements the desired shape must be defined in a Shape Table. This table consists of a series of integer values describing the shape desired. Figure 13-1 shows a Shape Table form where the user enters information called M-values about the shape to be generated. The initial point where you start drawing the shape is the previous X, Y coordinates from a graphics statement. The shape required should be drawn on graph paper, one dot per square starting at a particular X, Y coordinates. This coordinate is the last one specified in a graphics statement before the FGDRAW is given. The shape can be encoded moving in increments of one pixel in the up, down, right or left direction. If the point you are at is to be set (lit), it is specified in the next movement description (see below). Hence two types of movement are possible: a move without setting the previous point and a move that also sets the previous point.

STEP 1: Enter M-values* In sequence in M(n) and M(n+1) columns below to obtain Num(m) integer values.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Num(m) | = | M(n) | x 16 | + | M(n+1) | = | _____ |
| Num(1) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(2) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(3) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(4) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(5) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(6) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(7) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(8) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(9) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(10) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(11) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(12) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(13) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(14) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(15) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(16) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(17) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(18) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(19) | = | _____ | x 16 | + | _____ | = | _____ |
| Num(20) | = | _____ | x 16 | + | _____ | = | _____ |
| . | | | | | | | |
| . | | | | | | | |
| . | | | | | | | |
| Num(m) | = | _____ | x 16 | + | _____ | = | _____ |

If more space is needed use another form, keeping numbers in sequence

STEP 2: Create Shape Table

A$=CHR$ (Num(1),Num(2),..., Num(m))

A$=CHR$(___,___,___,___,___,___,___,___,___,___,___,___,...)

STEP 3: Enter A$ in your program

LEGEND

*M-Values

| Direction of Move | Move Only | Set Previous Pixel & Move |
|---|---|---|
| Up | 0 | 4 |
| Right | 1 | 5 |
| Down | 2 | 6 |
| Left | 3 | 7 |

| Num(m) | Integer Values |
|---|---|
| Num(1) | = M(0)x16+M(1) |
| Num(2) | = M(1)x16+M(2) |
| | . |
| | . |
| | . |
| Num(m) | = M(n)x16+M(n+1) |
| m | = 1,2,... |
| n | = 0,1,2... |

Figure 13-1. Form To Create Shape Table

The following M-values are available to accomplish this task:

| Direction of Move | M-Values: Move only | M-Values: Set Previous Pixel & Move |
|---|---|---|
| Up | 0 | 4 |
| Right | 1 | 5 |
| Down | 2 | 6 |
| Left | 3 | 7 |

The corresponding M-values for the shape to be drawn are entered on the form on Figure 13-1. Two are entered per line and multiplied and added as specified.

Suppose, for example, the following shape is to be encoded where S represents the previous point.

| | | |
|---|---|---|
| | X | |
| X | | X |
| X | | X |
| S | X | |

This shape is composed of the following M-values:

M(0) = 0 - Move up from point S
M(1) = 4 - Set previous Pixel & move up
---
M(2) = 4 - Set previous Pixel & move up
M(3) = 1 - Move right
---
M(4) = 5 - Set previous Pixel & move right
M(5) = 2 - Move down
---
M(6) = 6 - Set previous Pixel & move down
M(7) = 6 - Set previous Pixel & move down
---

M(8) = 3 - Move left
M(9) = 7 - Set previous Pixel & move left

These M-values are entered in the form on Figure 13-1 in consecutive order to produce the following integer values.

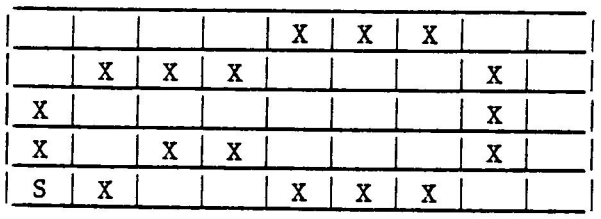
Num(m) = M(n) X 16 + M(n+1)
Num(1) = 0 X 16 + 4 = 4
Num(2) = 4 X 16 + 1 = 65
Num(3) = 5 X 16 + 2 = 82
Num(4) = 6 X 16 + 6 = 102
Num(5) = 3 X 16 + 7 = 55

These integer values can then be stored in the shape table and referred to a particular string variable using the CHR$ function, as follows:

```
A$ = CHR$ (Num(1),Num(2), .........)
   = CHR$ (4,65,82,102,55)
```

The statement "FGDRAW A$" draws the shape.

Example: Generate and display the following shape starting at the lower left corner of the screen and fill the shape with green:




STEP A - Generate the Shape Table:

```
M(0) = 0   - Move up
M(1) = 4   - Set previous pixel & move up
----------
M(2) = 4   - Set previous pixel & move up
M(3) = 5   - Move right
----------
M(4) = 5   - Set previous pixel & move right
M(5) = 5   - Set previous pixel & move right
----------
M(6) = 4   - Set previous pixel & move up
M(7) = 1   - Move right
----------
M(8) = 5   - Set previous pixel & move right
M(9) = 5   - Set previous pixel & move right
----------
M(10) = 5  - Set previous pixel & move down
M(11) = 2  - Move down
----------
M(12) = 6  - Set previous pixel & move down
M(13) = 6  - Set previous pixel & move down
----------
M(14) = 6  - Set previous pixel & move down
M(15) = 3  - Move left
----------
M(16) = 7  - Set previous pixel & move left
M(17) = 7  - Set previous pixel & move left
-----------
```

M(18) = 4 - Set previous pixel & move up
M(19) = 3 - Set previous pixel & move left
---
M(20) = 7 - Set previous pixel & move left
M(21) = 6 - Set previous pixel & move down
---
M(22) = 3 - Move left
M(23) = 7 - Set previous pixel & move left

STEP B - Enter the above values on the form in Figure 13-1 as follows:

Step 1: Enter M-values*

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Num(m) | = | M(n) | X | 16 | + | M(n+1) | = | |
| Num(1) | = | 0 | X | 16 | + | 4 | = | 4 |
| Num(2) | = | 4 | X | 16 | + | 1 | = | 65 |
| Num(3) | = | 5 | X | 16 | + | 5 | = | 85 |
| Num(4) | = | 4 | X | 16 | + | 1 | = | 65 |
| Num(5) | = | 5 | X | 16 | + | 5 | = | 85 |
| Num(6) | = | 5 | X | 16 | + | 2 | = | 82 |
| Num(7) | = | 6 | X | 16 | + | 6 | = | 102 |
| Num(8) | = | 6 | X | 16 | + | 3 | = | 99 |
| Num(9) | = | 7 | X | 16 | + | 7 | = | 119 |
| Num(10) | = | 4 | X | 16 | + | 3 | = | 67 |
| Num(11) | = | 7 | X | 16 | + | 6 | = | 118 |
| Num(12) | = | 3 | X | 16 | + | 7 | = | 55 |
| Num(13) | = | | X | 16 | + | | = | |

Step 2: Create Shape Table

```
A$=CHR$(Num(1),Num(2) , ...., Num(m))
A$=CHR$(4,65,85,65,85,82,102,99,119,67,
       118,55)
```

Step C - Enter Shape Table in Program

```
LIST¶
 5 ;CHR$(12) ! CLEAR LO-RES SCREEN
10 FGCTL 2
20 FGPOINT 0,0,0
30 FGFILL 239,239
40 FGPOINT 0,0,2
50 A$=CHR$(4,65,85,65,85,82,102,99,119,67,118,55)
60 FGDRAW   A$
70 FGPAINT 1,1,2  ! PAINT SHAPE
75 ! PAINT REMAINING AREA NOT COVERED BY PREVIOUS PAINT
80 FGPAINT 6,1,2
BASIC
```

## 13.7 FGERASE STATEMENT

**Function:** Sets all elements of the shape being drawn to the background color.

**Mode:** Program/Direct

**Format:** FGERASE <string var 1>

**Arguments:** String var 1 contains a shape table.

**Use:** FGERASE is used to erase a particular shape from the display when parts of the display are to be saved. This statement resets all the points in the shape to the display's background color. The specified shape is specified via its shape table contained in a string variable. (See FGDRAW Section 13.6.)

**Example:**

```
LIST¶
 5 ;CHR$(12) ! CLEAR LO-RES SCREEN
10 FGCTL 2
20 FGPOINT 0,0,0
30 FGFILL 239,239
40 FGPOINT 0,0,2
50 A$=CHR$(4,65,85,65,85,82,102,99,119,67,118,55)
60 FGDRAW A$
70 FGPAINT 1,1,2 ! PAINT SPACE
80 FGPAINT 6,1,2 ! PAINT REMAINING AREA NOT
                   COVERED BY PREVIOUS PAINT
90 FGPOINT 0,0,0 ! SET COLOR FOR ERASE
100 FGERASE   A$
BASIC
```

## 13.8 FGFILL STATEMENT

**Function:** Fills a rectangle from the previous position to the position indicated by the coordinates (x,y).

**Mode:** Program/Direct

**Format:**
1. FGFILL x,y[,color1]
2. FGFILL x,y, 256 * <color2> + <color1>

**Arguments:** x,y are the coordinates (0-239, 0-239) of the pixel directly opposite the previous position which together form the opposite corners in a rectangle. It can be an integer constant or variable.

Color1 specifies which one of the two (animation mode) or four-color choices the rectangle will be colored. It can take on values between 0 and 3. If omitted, the pixel's color will be the previous color in effect.

Color2 is optional and specifies that if the previous pixels color was color2 (0 to 3) it will be protected and not overwritten by color1. This is useful in the animation mode where one picture is shown while the other is updated.

**Use:** FGFILL can be used to fill a particular rectangular area on the screen or to clear part or all of the screen.

Example:

```
LIST¶
10  ;CHR$(12) ! CLEARS THE SCREEN
15  FGCTL 3 ! COLORS BL,RD,GR + BL
20  FGPOINT 0,0,1 ! SETS PIXEL 0,0 IN RED
30  FGFILL 239,239 ! FILLS SCREEN WITH RED
40  FGFILL 0,0,0  ! CLEARS SCREEN
BASIC
```

## 13.9 FGLINE STATEMENT

**Function:** Draws a line from the previous position to the position indicated by the specified coordinates.

**Mode:** Program/Direct

**Format:**

1. FGLINE x,y[,color1]
2. FGLINE x,y,256 * <color2> + <color1>

**Arguments:** x,y are the coordinates of the pixel to where the line will be drawn.

Color1 specifies which one of the two (animation mode) or four-color choices the line will be colored. It can take on values between 0 and 3. If omitted, the pixels color will be the previous color in effect.

Color2 is optional and specifies that if the previous pixel's color was color2 (0 to 3) it will be protected and not overwritten by color1. This is useful in the animation mode where one picture is shown while the other is updated.

**Use:** This statement sets all pixels to fill the line being drawn from previous point to the specified x,y position.

Example:

Ex. 1

```
LIST¶
10  FGPOINT 0,0,0 : REM SETS PIXEL 0,0 IN COLOR 0
15  PRINT CHR$(12) : REM CLEARS THE DISPLAY STORAGE
20  FGFILL 239,239 : REM CLEARS THE HIGH RES STORAGE
30  FGCTL 3 : REM SELECTS COLORS BK, R, GR, B + TEXT
35  REM DRAW A SQUARE
40  FGPOINT 20,20,2 : REM SETS PIXEL 20,20 IN COLOR
45  REM 2 (GR)
50  FGLINE 220,20 : REM DRAWS A LINE TO 220,20 IN
55  REM COLOR 2
60  FGLINE 220,220,3: REM DRAWS A LINE TO 220,220
    IN COLOR 3(B)
70  FGLINE 20,220,2 : REM DRAWS A LINE TO 20,220
75  REM IN COLOR 2 (GR)
80  FGLINE 20,20 : REM DRAWS A LINE TO 20,20 IN
85  REM COLOR 2
90  PRINT CUR(12,15); CYA DBLE "SQUARE";
100 END
BASIC
```

Example:

Ex. 2

```
LIST¶
  2  ;CHR$(12) ! CLEAR LOW RES SCREEN
  5  REM ** ANIMATION*
 10  REM ** THIS PROGRAM DRAWS A BLUE BAR
 20  REM ** ON LEFT SIDE OF SCREEN AND
 30  REM ** MOVES IT FROM X=1 TO X=230 POSITION
 40  FGPOINT 0,0,0
 50  FGFILL 239,239
 60  FGCTL  109
 70  C=1 : X=1 : Y=1
 80  Z=FNDRAW(C)
 90    C=2
100  G=FNDRAW(C)
110  R=FNSHOW(C)
120  S=FNERASE(C)
130  IF C=1 THEN C=2 ELSE C=1
140  IF X < 230 THEN GOTO 100
150  DEF  FNDRAW (C)
160    FGPOINT X,Y,C
170    FGLINE X,Y + 100, C
180    X=X + 2
190    RETURN 0
200  FNEND
210  DEF FNERASE(C)
220  FGPOINT X-4,Y,0
230  FGLINE X-4,Y+100,0
240  RETURN 0
250  FNEND
260  DEF FNSHOW(C)
270    IF C=1 THEN FGCTL 108 ELSE FGCTL 109
280    RETURN 0
290  FNEND
300  END
BASIC
```

## 13.10 FGPAINT STATEMENT

**Function:** Fills a closed area with a specified color.

**Mode:** Program/Direct

**Format:**

1. FGPAINT x,y[,color1]
2. FGPAINT x,y,256 * <color2> + <color1>

**Arguments:** x,y are the coordinates (0-239, 0-239) of the pixel inside the closed area to be filled with a specified color.

Color1 specifies which one of the two (animation mode) or four-color choices the closed area will be colored. It can take on values between 0 and 3. If omitted, the pixel's color will be the previous color in effect.

Color2 is optional and specifies that if the previous pixel's color was color2 (0 to 3) it will be protected and not overwritten by color1. This is useful in the animation mode where one picture is shown while the other is updated.

**Use:** In order to paint an object or the area around an object, a point x,y must be specified whose coordinates represent a certain spot on the console. This point determines where the painting will begin and exactly what area will be painted during the execution of a single instruction. Thus, the position of the point within the object being pointed or outside the object (i.e., when coloring the area around the object) is very important with respect to the performance of the FGPAINT statement.

Painting takes place in the following manner:

Step 1: Starting at (x,y), an upward traversal is made until the end of screen or a new color is found.

Step 2: Then, coloring takes place horizontally from right to left, filling in the area between the end of screen and the object, or within the object itself, until the point x,y is reached.

Step 3: A downward traversal is then made until the end of screen or a new color is found.

Step 4: Once again, coloring takes place on a horizontal basis, painting the area between the object and the end of screen or between the object's boundaries.

Note: Remember, however, that depending upon the location of the point with respect to the area to be painted, some of the area may or may not be colored as desired. The location of the point must then be changed to accommodate for such an occurrence. With practice, you will find that most objects may be colored with one or two FGPAINT statements.

Example:

```
LIST¶
10  ! DRAW A CIRCLE WITHOUT USING FGCIRCLE
20  EXTEND
30  C=1
40  ; CHR$(12) ! CLEARS THE SCREEN
50  FGPOINT 0,0,0 ! SETS PIXEL 0,0 IN COLOR 0
60  FGFILL 239,239 ! FILLS THE SCREEN WITH COLOR 0
65  FGPOINT 0,0,0
70  FGCTL 7 ! SELECTS A COLOR COMBINATION
80  ORIGIN=119
90  RADIUS=103
100 COLOR=1
110 FOR XPOSITION=-RADIUS TO RADIUS STEP .2
120   HEIGHT=SQR(RADIUS*RADIUS-XPOSITION*XPOSITION)*C
130   FGPOINT XPOSITION+ORIGIN,ORIGIN-HEIGHT,COLOR
140   FGPOINT XPOSITION+ORIGIN,ORIGIN+HEIGHT
150 NEXT XPOSITION
160 FOR YPOSITION=-RADIUS TO RADIUS STEP .2
170   WIDTH=SQR(RADIUS*RADIUS - YPOSITION*YPOSITION)
180   FGPOINT ORIGIN-WIDTH,YPOSITION+ORIGIN
190   FGPOINT ORIGIN+WIDTH,YPOSITION+ORIGIN
200 NEXT YPOSITION
210 FGPAINT ORIGIN,ORIGIN !PAINTS CIRCLE IN RED
220 END
BASIC
```

## 13.11 FGPOINT FUNCTION

**Function:** Returns the color number of a specified pixel.

**Mode:** Program/Direct

**Format:** FGPOINT (x,y)

**Arguments:** x,y are the coordinates of the pixel being interrogated for color number.

**Example:**

```
LIST¶
10  FGTCL 3
20  FGPOINT 0,0,3
40  FGLINE 0,10
50  ;FGPOINT (0,5)
RUN
3
BASIC
```

## 13.12 FGPOINT STATEMENT

**Function:** Turns on a particular pixel in the specified or previous color.

**Mode:** Program/Direct

**Format:**
1. FGPOINT x,y [,color1]
2. FGPOINT x,y,256*<color2> + <color1>

**Arguments:** x,y are the coordinates of the pixel to be turned on (on line x (0-239) in position y (0 to 239)).

Color1 specifies which one of the two (animation mode) or four-color choices the pixel will be colored. It can take on values between 0 and 3. If omitted, the pixel's color will be the previous color in effect.

Color2 is optional and specifies that if the previous pixel's color was color 2 (0 to 3) it will be protected and not overwritten by color1. This is useful in the animation mode where one picture is shown while the other is updated.

**Use:** FGPOINT is used to initially turn on a particular pixel. Other high resolution statements can then be given to draw lines, rectangles, and fill areas starting at this point.

Examples:

Ex. 1

```
LIST¶
 2 ;CHR$(12) ! CLEAR LOW RES SCREEN
 5 FGCTL 3 ! COLORS BK,RD,GR, + BL
10 FGPOINT 5,5,2 ! SETS PIXEL 5,5 IN GREEN
20 FGLINE 15,15 ! DRAWS LINE IN GREEN
30 FGPOINT 15,10,256*2+1 !SETS PIXEL 10,10 IN
35 ! RED ONLY IF GREEN WASN'T PREVIOUSLY PRESENT
BASIC
```

Ex. 2

In this example two points ((0,0) and (0,200)) are protected while a second picture is drawn with color black.

```
LIST¶
 5  ;CHR$(12) ! CLEAR LOW RES SCREEN
10  FGCTL    200  ! COLORS BK, R, BK, R
20  FGPOINT  0,0,0
30  FGFILL   239,239
40  FGPOINT  0,0,1
50  FGLINE   100,100
60  FGLINE   200,0
70  ! PROTECT (0,0) FROM BEING OVERWRITTEN BY BLACK
80  FGPOINT  0,0,256*1+2
90  FGLINE   100,105,256*1+2
100 ! PROTECT (200,0) FROM BEING OVERWRITTEN
105 ! BY BLACK
110 FGLINE   200,0,256*1+2
120 FGCTL    201 ! COLORS BK, BK, R, R
BASIC
.
.
```

NOTE: List command must be entered after executing this program to get back to BASIC.

## 13.13 FGPUT STATEMENT

**Function:** Restores to high resolution memory the contents of a rectangle specified by a string variable via an FGGET statement.

**Mode:** Program/Direct.

**Format:** FGPUT <string var>

**Arguments:** String var contains header and graphics data for the rectangle copied by the FGCOPY statement.

**Use:** The high resolution graphics rectangular data is described by an 8-byte header and succeeding graphics data. FGPUT uses this data to restore the rectangle, stored by FGGET, starting at the previous x,y location. Parts of the 8-byte header can be changed by the user (see example) to alter the shape and relative position of the restored rectangle. The header's contents have been described in detail under FGGET and are summarized below:

| Byte | AS |
|---|---|
| 1 | xpos |
| 2 | ypos |
| 3 | xsize |
| 4 | ysize |
| 5 | xst |
| 6 | yst |
| 7 | xcant |
| 8 | ycant |
| | Graphics data |

Suppose, for example, the shape of a rectangle is to be reduced to half in both length and width. Hence, the 7th and 8th bytes (xcnt, ycnt) in the header have to be changed. This can be done using the MID$ function as follows (assume xcnt=200, ycnt=100):

```
MID$(A$,7,2) = CHR$(100,50)
```

Example:

```
LIST¶
 2 ;CHR$(12) ! CLEAR LOW RES SCREEN
 5 ! DEFINE COLOR, LOCATION AND SIZE OF RECTANGLE
10 FGCTL 2
20 FGPOINT 0,0,0
30 FGFILL 239,239
40 FGPOINT 10,10,1
50 FGFILL 90,90
60 FGPOINT 11,11,2
70 FGFILL 89,89
80 ! COLLECT THE RECTANGLE IN A$
90 FGPOINT 10,10,1
100 A$ = FGGET (90,90)
110 ! PUT RECT FROM 100,100
120 FGPOINT 100,100
130 FGPUT A$
140 ! REDUCE SIZE OF RECT AND PUT FROM
150 ! POST. 120,10
155 ! 7TH AND 8TH BYTE ARE X AND Y PIXEL
156 ! COUNTS. ORIGINALLY 90,80 CHANGE TO 45,45.
160 MID$(A$,7,2)=CHR$(45,45)
163 ! CHANGE THE REL. POS. FROM U.L. CORNER
164 ! TO OLD POS. ORIGINALLY YPOS=90 CHANGE TO 45
166 MID$ (A$,2,1)=CHR$(45)
170 FGPOINT 120,10,0
180 FGPUT A$
182 ! PUT OFF THE SCREEN
190 FGPOINT 230,10,0
200 FGPUT A$
BASIC
```

## 13.14 FGROT STATEMENT

**Function:** Specifies the degree of rotation to be applied to the shape(s) displayed by subsequent executions of FGDRAW and FGERASE.

**Mode:** Program/Direct.

**Format:** FGROT <number>

**Arguments:** Number is an integer specifying the degree of rotation to be applied to the shape being displayed, as follows:

| Number | Meaning |
|---|---|
| 0 | No rotation (initial value assumed) |
| 1 | 45° |
| 2 | 90° |
| 3 | 135° |
| 4 | 180° |
| 5 | 225° |
| 6 | 270° |
| 7 | 315° |

**Use:** Use rotation value of 0 when no rotation is desired. An initial value of 0 will be assumed unless a FGROT statement was previosly executed. The FGROT statement which is to rotate the shape must appear in a program before the FGDRAW statement.

**Example:**

```
LIST¶
 2 FGSCALE 1,1 :FGROT 0  ! RESET SCALE AND ROTATION
 5 ;CHR$(12) ! CLEAR LOW RES SCREEN
10 FGCTL 2
20 FGPOINT 0,0,0
30 FGFILL 239,239
40 FGPOINT 120,120,2
50 A$=CHR$(4,65,85,65,85,82,102,99,119,67,118,55)
55 FGROT   2 ! ROTATE SHAPE 90 DEGREES
60 FGDRAW   A$
BASIC
```

## 13.15 FGSCALE STATEMENT

**Function:** Scales either or both the x or y coordinate of the shape to be displayed by the FGDRAW or FGERASE statement.

**Mode:** Program/Direct.

**Format:** FGSCALE <x,y>

**Arguments:** x is a nonzero positive integer and specifies the multiplier of the x-dimension of the shape(s) to be displayed.

y is the corresponding multiplier for the y-dimension of the shape(s).

**Use:** Use scale values of 1 when no scaling is desired. An initial value of 1 will be assumed unless a FGSCALE statement was exacuted previosly. The FGSCALE statement which is to scale the shape must appear in a program before the FGDRAW statement.

**Example:**

```
LIST¶
 3 FGSCALE 1,1 ! RESET SCALE TO NORMAL
 4 FGROT 0 ! RESET ROTATION TO 0
 5 CHR$(12) ! CLEAR LOW RES SCREEN
10 FGCTL 2
20 FGPOINT 0,0,0
30 FGFILL 239,239
40 FGPOINT 0,0,2
50 A$=CHR$(4,65,85,85,82,102,99,119,67,118,55)
55 FGSCALE   2,1  ! SCALE SHAPE 2 X IN X DIRECTION
60 FGDRAW   A$
BASIC
```

# SECTION 14
# ADVANCED PROGRAMMING

# SECTION 14
# ADVANCED PROGRAMMING

## 14.1 INTRODUCTION

This section contains information that should only be applied by user's who have a complete understanding of Monroe BASIC and the Monroe Operating System. Subjects discussed here include advanced statements and functions, file creation and access methods.

## 14.2 ADVANCED STATEMENTS AND FUNCTIONS

This section contains Monroe BASIC statements and functions which are to be used for advanced programming. The user is cautioned that if certain of these statements are used incorrectly, program execution or parts of the operating system may be inadvertently destroyed. The advanced programming statements and functions are summarized in Table 14-1.

Table 14-1. Advanced Programming Statements and Functions

| Statement or Function | Use |
|---|---|
| CALL | Calls an Assembler Program. |
| CVT | Permits floating-point and integer values to be represented in binary in ASCII I/O files. |
| INP | Returns value of data from in-port specified. |
| ISAM CREATE PROCEDURE | Creates an ISAM index file and specifies the associated data file. |
| ISAM DELETE | Deletes a record from an ISAM index file. |
| ISAM OPEN | Opens an ISAM index file and its associated data file. |
| ISAM READ | Accesses an ISAM data file. |
| ISAM UPDATE | Modifies an existing record in the data file associated with an ISAM index file. |

Table 14-1. Advanced Programming Statements and Functions (Cont.)

| Statement or Function | Use |
|---|---|
| ISAM WRITE | Enters a new record into the data file and updates all indices in the index file. |
| OPEN | Opens file in Record or Byte I/O mode. |
| OUT | Sends data to the out-port specified. |
| PEEK | Returns memory contents of a specified address. |
| PEEK2 | Reads the contents of two bytes. |
| POKE | Changes or loads a value into specified address. |
| POSIT | Positions file pointer to record or byte position desired or returns the current position of the pointer. |
| PREPARE | Opens (and allocates) a new file in byte I/O mode. |
| SVC | Communicates with operating system to perform special functions. |
| SWAP% | Transposes first and second bytes of an integer. |
| SYS(A) | Provides essential system information. |
| VAROOT | Returns starting address of a table containing data about a variable. |
| VARPTR | Returns starting address of where a variable is stored. |

## CALL Function

**Function:** Calls an Assembly program and returns the contents of CPU register HL.

**Mode:** Direct/Program.

**Format:** CALL(A%[,D%])

**Arguments:** A% is an integer holding the address of the machine code being called.

D% is optional integer parameter which will be placed in CPU Register DE of the processor at the call.

**Note:** The assembly routine should always return to Monroe BASIC by executing a return instruction.

**Result:** The contents of register HL (integer) will be returned as the value of the function.

**Example:**

```
LIST¶
 5  ! DEFINE ADDRESS WHERE ASSEMBLY ROUTINE STARTS.
10  A%=1234%
15  ! DEFINE THE PARAMETER WHICH WILL BE TRANSFERRED
17  ! TO ROUTINE
20  D%=ASCII("A")
25  ! CALL THE ASSEMBLY ROUTINE AND PUT THE RETURNED
27  ! RESULT IN H%
30  H%=CALL(A%,D%)
40  END
```

**Caution:** This function is machine-oriented and should only be used for advanced programming. CALL can destroy a program execution if used erroneously.

## CVT Conversion Functions

CVT Conversion Functions are provided to permit floating-point and integer values to be represented in binary in ASCII I/O files. These functions are:

| Function | Form | Description |
|---|---|---|
| CVT%S(I%) | A$=CVT%S(I%) | Maps an integer into a two-character string. |
| CVTS%(AS) | I%=CVTS%(AS) | Maps the first two characters of a string into an integer. The string must have at least two characters. |
| CVTFS(X) | AS=CVTF$(X) | Maps a floating-point number into a four- or eight-character string (depending upon whether Single or Double precision is used). |
| CVT$F(AS) | X=CVT$F(AS) | Maps the first four or eight characters (depending upon whether Single or Double precision is used) of a string into a floating-point number. The string must have enough characters; otherwise, wrong results will be returned. |

The above functions do not affect the value of the data, but rather its storage format. Each character in a string requires one byte of storage (8 bits); hence, characters may assume (decimal) values from 0 through 255 and no others. A 16-bit quantity can be defined as either an integer or a two-character string; two-word floating point numbers can equally be defined as four-character strings.

The four CVT Conversion Functions are described in detail in subsequent paragraphs.

## CVT%S Function

**Function:** Returns a two-character string representation of an integer.

**Mode:** Program/direct

**Format:** CVT%S(<variable>)

**Arguments:** Variable can be an integer constant, integer variable or a subscripted variable.

**Use:** This function permits dense packing of data in records. For example, any integer value between -32768 and 32767 can be packed in a record in two characters. This would only be true for integers between -9 and 99 if the data was stored as ASCII characters.

**Example:**

```
LIST¶
 10    RANDOMIZE
 20  DIM A$(100%)
 30  ! GENERATE 10 FIVE-DIGIT RANDOM INTEGER NUMBERS
 40  !
 41  ; "*** INTEGERS GENERATED ***"
 42  ;
 50  FOR I%=1% to 10%
 70    A%(I%)=INT(RND*32767%)
 75    ; A%(I%)
 80  NEXT I%
 81  ;
 90  !
100  ! THE INTEGERS ABOVE CAN BE STORED INTO A FILE IN
105  ! TWO WAYS
110  !
120  ! 1. USING THE PUT AND NUM$ STATEMENTS
130  ! ...THE SIZE OF FILEA WILL BE 50 BYTES...
```

```
131  !
140  PREPARE "FILEA" AS FILE 1
150  FOR I%=1% TO 10%
157    S$=NUM$(A%(I%))
158    S$=S$+SPACE$(5%-LEN(S$))
160    PUT #1%,S$
170  NEXT I%
180  CLOSE 1%
181  !
185  !
190  ! 2. USING THE PUT AND CVT%$ STATEMENTS
200  ! ...THE SIZE OF FILEB WILL BE 2 x 10 = 20 BYTES
201  ! ...FILEB WILL BE PACKED FROM 50 BYTES TO 20
202  ! ...BYTES BY USING CVT%$.
203  !
210  PREPARE "FILEB" AS FILE 2
220  FOR I%=1% TO 10%
230    PUT #2% CVT%$(A%(I%))
240  NEXT I%
250  CLOSE 2%
260  !
270  END
BASIC
```

## CVT$% Function

**Function:** Returns the integer representation of the first two characters of a binary string (having a least two characters).

**Mode:** Program/direct

**Format:** CVT$%(<string>)

**Arguments:** String is any string variable or constant but only the first two characters will be converted at a time.

**Use:** The CVT$% function provides the means to speed the processing of a large amount of packed data within a file. Converting the internal binary representation to an ASCII string is a less time-consuming process with CVT$% than the NUM$ function.

**Example:**

```
LIST¶
270  ! THIS PROGRAM READS THE INTEGERS FROM THE FILES
275  ! CREATED FOR PREVIOUS CVT%$ EXAMPLE
280  !
290  ! 1. FROM FILEA
300  DIM B% (10%)
310  OPEN "FILEA" AS FILE 1
320  FOR J%=1% TO 10%
321    GET #1%,B$ COUNT 5
331    B% (J%)=VAL(B$)
350  NEXT J%
351  ;
355  CLOSE 1%
360  !
370  ! 2. FROM FILEB BY USING CVT$%
380  !
390  OPEN "FILEB" AS FILE 2
```

```
400   FOR J%=1% TO 10%
430     GET #2%,B$ COUNT 2
440     B%(J%)=CVT$%(B$)
450   NEXT J%
460   CLOSE 2%
470   END
BASIC
```

## CVTFS Function

Function: Returns the four- or eight-character string representation of a floating point number depending on whether Single or Double precision was in effect.

Mode: Program/direct

Format: CVTF$(<n>)

Arguments: n is a Single or Double precision floating-point number.

Use: This function permits dense packing of floating point data in records. For example, any floating point number between $2.93874 \times 10^{-39}$ through $1.70141 \times 10^{38}$ (single precision) can be stored in a four-character string and between $2.938735877055719 \times 10^{-39}$ through $1.70141183460492 \times 10^{-38}$ in an eight-character string (double precision).

Example:
```
LIST¶
   2  !  THIS PROGRAM STORES A FLOATING POINT ARRAY
   4  ! ON A DISC FILE IN A COMPACT FASHION
  10  DIM A(100)
1000  PREPARE "FIL" AS FILE 1%
1010  FOR I% = 1% to 100%
1020  PUT #1%, CVTF$(A(I%))
1030  NEXT I%
1040  CLOSE 1%
BASIC
```

## CVT$F Function

**Function:** Returns the floating point number representation of the first four- (Single precision) or eight-character (Double precision) of a string.

**Mode:** Program/direct

**Format:** CVT$F(<string>)

**Arguments:** String is any string variable or constant having at least four characters (Single precision) or eight characters (Double precision). Unknown results will be returned if the string length is less than four (Single) or eight (Double).

**Use:** The CVT$F function provides the means to speed the processing of a large amount of packed data within a file. Converting the internal binary representation to an ASCII string is a less time-consuming process with CVTS% than the NUM$ function.

**Example:**

```
LIST¶
    2  ! THIS PROGRAM READS BACK THE ARRAY
    3  ! CREATED BY EXAMPLE FOR CVTF$
    4  ! LEN (CVTF$(0)) IS USED TO DETERMINE IF
    5  ! SINGLE OR DOUBLE PRECISION IS USED
   10  DIM A(100):L%=LEN(CVTF$(0))
 2000  OPEN "FIL" AS FILE 1%
 2010  FOR I% = 1% to 100%
 2020  GET #1 AS COUNT L% : A(I%) = CVT$F(A$)
 2030  NEXT I%
 2040  CLOSE 1%
BASIC
```

## INP Function

**Function:** Returns the value of data from the in-port specified.

**Mode:** Direct/Program

**Format:** INP(<port>)

**Argument:** All port numbers are specified in decimal.

**Note:** See Appendix D for Port numbers. The user should be familiar with I/O Device Programming. Refer to Z80-SIO,-CTC-, and -PIO Technical Manuals for details.

**Example:**

```
LIST¶
10  !
20  !  SIMPLE EXAMPLE SHOWING HOW TO READ A CHARACTER
25  ! FROM THE SIO.
30  ! CHECK FLAG IN COMMAND PORT AND
35  ! JUMP IF NO CHARACTER
40  IF (1% AND INP(10%*16%+5%)) = 0% THEN 110
45  ! READ CHARACTER FROM DATA PORT.
50  A%=INP(10%*16%+4%)
60  !
65  ! CAUTION
70  !! THIS ROUTINE WILL ONLY WORK UNDER THE
71  ! CONDITIONS
75  ! THAT NO OTHER TASK OR INTERRUPT ROUTINE READ OUT
80  ! THE DATA FROM DATA PORT BETWEEN LINES 40 AND 50.
85  ! E.G. THERE COULD BE OTHER INTERRUPT ROUTINES
90  ! WHICH ARE ENABLED IN THE SYSTEM USING THE SIO
95  ! DURING EXECUTING OF LINE 40 AND 50
110  END
BASIC
```

## ISAM Create Procedure

Function: Allocates and creates an ISAM Index file and its associated Data file.

Format: CREINDEX

NOTE: CREINDEX is a task utility program. It is run at the operating system level.

Use: To create and allocate ISAM files requires the execution of Utility Program CREINDEX. Refer to the 8800 Series "Monroe Utility Programs Programmer's Reference Manual - Part 1" for important information about ISAM files and procedure instructions for CREINDEX.

When this program is executed it prompts the user as follows:

```
Enter name of index file?_____________
Enter name of data file?_____________
Enter record length?_____________
Enter key start position?_____________
Enter key type (B,A,I,F or D)?_____________
Ascending or Descending sequence (A/D)?_____________
Are duplicate key values allowed? (Y or N)_____________
Are there any more indices? (Y or N)?_____________
Is information correct (Y or N)?_____________
```

If there are any more indices, the user is returned to the first query inputting the name of the index file, the name of the data file, and so on, until all indices have been entered. Then a table is output to the console summarizing all of the information entered during the session.

```
Information correct (Y or N)?_____________
Would you like a copy on the printer (Y or N)?_____
```

Example: This example illustrates how an index and a data file are allocated, created and then built. Three keys are specified: Name, Address and Phone Number.

```
- CREINDEX¶

*** CREATE ISAM FILES VER. 3.02 ***
Enter name of index file? IFILE¶
Preallocate space (Y or N? N¶
Enter name of data file? DATA¶
Preallocate space (Y or N)? N¶
Enter record length? 37¶
Enter name of index? NAME¶
Enter key start position? 1¶
Enter key length? 10¶
Enter key type (B,A,I,F or D)? A¶
Ascending or Descending sequence (A/D)? A¶
Are duplicate key values allowed? (Y or N)? Y¶
Are there any more indices? (Y or N)? Y¶
Enter name of index? TOWN¶
Enter key start position? 20¶
Enter key length? 10¶
Enter key type (B,A, I, F or D)? A¶
Ascending or Descending sequence (A/D)? A¶
Are duplicate key values allowed? (Y or N)? Y¶
Are there any more indices? (Y or N)? Y¶
Enter name of index? PHONE¶
Enter key start positions? 30¶
Enter key length? 7¶
Enter key type (B,A,I,F or D)? A¶
Ascending or Descending sequence (A/D)? A¶
Are duplicate key values allowed? (Y or N)? N¶
Are there any more indices? (Y or N)? N¶
```

The following output appears on the console:

```
***Create Isam Files*** Ver. P-3.02 yyyy-mm-dd/hh.mm.ss

   Data and Index File Information.

   Index File name:  IFILE
   Data File name :  DATA
   Record size    :  37
```

| | Filename | Reclgt | BlkSize | Allo blks |
|---|---|---|---|---|
| Index File: | IFILE | 256 | Default | Default |
| Data File: | DATA | 37 | Default | Default |

| Index No. | Index Name | Key Type | Sort Order | Dupl. | Key Start/Length |
|---|---|---|---|---|---|
| 1 | NAME | Ascii | Ascending | Yes | 1/10 |
| 2 | TOWN | Ascii | Ascending | Yes | 20/10 |
| 3 | PHONE | Ascii | Ascending | No | 30/7 |

```
Is information correct (Y or N)? Y¶
Would you like a copy on the printer (Y or N)? Y¶

Index file created!
Data file created!
Another run (y or n)? n¶
hh.mm.ss  End of Task 0
```

```
-BASIC¶

BASIC Rx-xx
BASIC
```

(The following program enters name, address and phone information into data file IFILE.)

```
 5 INPUT "ISAM INDEX FILE NAME? "J$¶
10 ISAM OPEN J$ AS FILE 1¶
20 INPUT "NAME ? "A$ : IF LEN (A$)>20 THEN A$=LEFT$(A$,20)¶
30 IF LEN(A$)=0 THEN STOP ELSE A$=A$+SPACE$(20-LEN(A$))¶
40 INPUT "TOWN ? "B$ : IF LEN(B$)>10 THEN B$=LEFT$(B$,10)¶
50 A$=A$+B$+SPACE$(10-LEN(B$))¶
60 INPUT "PHONE ? "B$ : IF LEN(B$)>10 THEN B$=LEFT$(B$,10)¶
70 A$=A$+SPACE$(10-LEN(B$))+B$¶
80 ; A$¶
90 ISAM WRITE #1,A$ : GOTO 20¶

RUN¶
ISAM INDEX FILE NAME ? IFILE¶
NAME ? J. JOHNSON¶
TOWN ? MORRISTOWN¶
NUMBER ? 540-7612¶
J. JOHNSON   MORRISTOWN    540-7612
NAME ? . . .
.
.
.
```

## ISAM DELETE Statement

**Function:** Removes a particular record from an ISAM index file.

**Mode:** Program/Direct.

**Format:** ISAM DELETE #<channel no.>,<stringvar>

**Arguments:** Channel no. corresponds to the internal number on which the file is opened. Valid channel numbers are 1 to 250.

Stringvar is a string variable which must be identical to the record last read on that <channel no.>.

**Use:** This statement removes the appropriate keys from a designated record in the ISAM file. The associated data record is not touched but subsequent access is not possible. Before an ISAM DELETE can be done the record must be ISAM READ.

**Example:**

```
10 ISAM OPEN "VOL:IFILE" AS FILE 1¶
20 ISAM READ #1,A$ INDEX "NAME" KEY "SMITH"¶
30 ISAM DELETE #1,A$¶
```

## ISAM OPEN Statement

**Function:** Opens both the index and data files for ISAM access on a disk.

**Mode:** Direct/Program.

**Format:** ISAM OPEN <string expression> AS FILE <channel no.>

**Arguments:** The string expression corresponds to an external file specification for the index file to be opened as specified in Section 1.3. The data file associated with this index file is automatically opened after the index file is specified.

The channel no. after AS FILE must have an integer value corresponding to the internal channel number on which the file is opened. Valid channel numbers are 1 to 250.

**Note:** ISAM OPEN when executed in a Monroe BASIC program loads an ISAM task into memory. If no more ISAM operations are to be performed, Utility program "KILLISAM" should be run to release the space occupied by the ISAM task.

**Use:** ISAM OPEN is the only method used to open an indexed file for ISAM access. Once this file is opened the data contained in the corresponding data file can be read, written, deleted and updated using the appropriate ISAM statements.

Note that one ISAM OPEN statement opens both the index and data files.

Examples:

Ex. 1

```
10 REM VOL:IFILE IS THE ISAM INDEX FILE ON VOLUME VOL¶
20 ISAM OPEN "VOL:INDEX" AS FILE 1¶
```

Ex. 2

```
10 REM PROGRAM PROMPTS FOR ISAM FILE NAME¶
20 INPUT "ISAM FILE NAME? "A$¶
30 ISAM OPEN AS AS FILE 1¶
```

## ISAM READ Statement

**Function:** Accesses by key or sequentially, records contained in the data file associated with an ISAM index file.

**Mode:** Direct/Program.

**Format:** ISAM READ #<channel no.>,<stringvar> [INDEX stringa] [[KEY stringb>][FIRST][LAST][NEXT][PREVIOUS]]

**Arguments:** The channel no. corresponds to the internal channel number on which the file is opened. Valid channel numbers are 1 to 250.

Stringvar is any legal string variable.

Stringa is either a string expression or string variable which defines the index that is to be used.

Stringb is either a string expression or string variable which defines the search argument within the index.

**Note:** The FIRST, LAST, NEXT, or PREVIOUS keyword can be used in place of [KEY stringb] to position the pointer to the first, last, next, or previous record in a particular index. That record can now be read without naming a particular key for stringb.

**Use:** The following rules are in effect for ISAM READ:

1. If the INDEX option (stringa) is missing or empty, the first is selected.

2. If the KEY option (stringb) is missing or empty, the first record by selected index is read.(See note above for exceptions to this rule.)

3. If both INDEX and KEY options are missing, a sequential read is performed. (See note above for exceptions to this rule.)

4. If it is the first read operation after ISAM OPEN, the first index is selected and the first record by that index is read.

5. The KEY string may be a substring of the record key. In this case, the first record (by key) that contains the key given is read.

6. If duplicate keys are present in the index, the first record that contains the key given is read.

Examples: The examples below illustrate the various ways ISAM READ can be used.

Ex. 1

```
10  ISAM OPEN "VOL:MASTS" AS FILE 1¶
20  ISAM READ #1, AS¶
    (Reads first index since INDEX option is missing)
```

Ex. 2

```
10  ISAM OPEN "VOL:MASTS" AS FILE 1¶
20  ISAM READ #1, AS INDEX "NAME"¶
or
10  I$="NAME"¶
20  ISAM READ #1,A$ INDEX I$¶
    (Reads first record by selected INDEX)
```

Ex. 3

```
10  ISAM OPEN "VOL:MASTS" AS FILE 1¶
20  ISAM READ #1,AS¶
30  FOR I%=1% TO N%¶
40    ISAM READ#1,AS¶
50    ; AS    ! PRINT THE RECORDS TO THE CONSOLE¶
60  NEXT I%¶
```

Ex. 4

```
10  OPEN "PR:" AS FILE 2¶
20  FOR I%=1% TO N%¶
30    ISAM READ #1,AS¶
40    ; #2, AS  ! PRINT THE RECORDS ON THE PRINTER¶
50  NEXT I%¶
    (Performs sequential read since both INDEX and
    KEY options are missing.)
```

Ex. 5

```
10  ISAM OPEN "VOL:IFILE" AS FILE 1
20  ISAM READ #1,A$ INDEX "NAME" KEY "SMITH"¶
    (Reads selected record by selected index -
    random access of a particular record.)
```

Ex. 6

```
10  ISAM OPEN "FIRST" AS FILE 2
20  ISAM READ #2, A$ INDEX "SSNUM" LAST
    (Reads last record by key using the
    "SSNUM" index.)
.
.
.
50  ISAM READ #2, B$ PREVIOUS
    (Reads the 2nd from the last record using
    the "SSNUM" key.)
```

## ISAM UPDATE Statement

**Function:** Alters an existing record in the data file and produces key changes to the index file when applicable.

**Mode:** Program/Direct.

**Format:** ISAM UPDATE #<channel no.>,<string1> TO <string2>

**Arguments:** Channel no. corresponds to the internal channel number on which the file is opened. Valid channel numbers are 1 to 250.

String1 is a string variable and must be identical to the record last read on that <channel no.>.

String2 is a string variable and will replace string1 in the data file. All changed indices will be updated when this replacement occurs.

**Use:** Before using ISAM UPDATE, the appropriate file and records must be ISAM opened and ISAM read. If a duplicate key occurs in an index where it is not allowed, that index will not be updated, and an error will result. For example, if the name SMITH was used as a key for record 50 and you wanted to change record 20's key to SMITH, an error would result. In order to keep the indices properly updated, an ISAM DELETE operation must be performed.

**Example:**

```
LIST¶
10 ISAM OPEN "VOL:IFILE" AS FILE 1
20 ISAM READ #1,A$ INDEX "NAME" KEY "SMITH"
30 B$="SMITH    NEW YORK    726-2677"
40 ISAM UPDATE #1,A$ TO B$
BASIC
```

## ISAM WRITE Statement

**Function:** Enters a new record into the data file associated with an ISAM index file and adds the new keys in the index file.

**Mode:** Program/Direct.

**Format:** ISAM WRITE #<channel no.>,<stringvar>

**Arguments:** The channel no. corresponds to the internal channel number on which the file is opened.Valid channel numbers are 1 to 250.

Stringvar is any legal string variable.

**Use:** The record is appended to the data file and all indices are updated. The record must contain information in all key fields. If a duplicate key occurs in an index where it is not allowed, that index will not be updated, and an error will result.

**Examples:** Ex. 1

```
10  ISAM OPEN "VOL:IFILE" AS FILE 1¶
20  ISAM WRITE #1,"SMITH    NEW YORK   632-3256"¶
```

Ex. 2

```
10  ISAM OPEN "VOL:IFILE" AS FILE 1¶
20  A$="SMITH    NEW YORK    632-3256"¶
30  ISAM WRITE #1,A$¶
```

OPEN Statement

Function: Opens a file for sequential or random access [illegible] file-structured [illegible] I/O [illegible] internal to the Monroe BASIC program. To open an indexed sequential file for ISAM access, refer to ISAM OPEN in this section.

Mode: Direct/Program

Format: OPEN <string expression> AS FILE <channel no>
[MODE a%+b%]

Arguments: String Expression specifies the name of the disk file to be opened.

The channel no. after AS FILE must have an integer value corresponding to the internal channel number on which the file is opened. Numbers 1 thourgh 250 are legal.

a% and b% are integers which determine mode I/O and Read/Write characteristics. The following codes are in effect:

| a% | Meaning |
|---|---|
| 0% | Record I/O is to be used. Record length must be specified in GET statement. |
| 128% | Physical I/O on disk sector level. If 128 is specified data in excess of 256 bytes will be written in blocks of 256 bytes. The first record will consist of 256 bytes, the second 256 bytes, etc. Hence the fastest wayu to write is in blocks of 256 bytes (i.e., 256, 512, 768, etc.) |
| 192% | Byte I/O is to be used (default). |

| b% | Meaning |
|---|---|
| 0% | Sharable Read Only (SRO) |
| 1% | Exclusive Read Only (ERO) |
| 2% | Shared Write Only (SWO) |
| 3% | Exclusive Write Only (EWO)<br>Normally when a file is opened Read and Writes take place from beginning of a file. However, this special mode will start at the end of file (append) if no positioning is done. |
| 4% | Shared Read/Write (SRW) |
| 5% | Sharable Read, Exclusive Write (SREW)<br>(default if b% = 192%) |
| 6% | Exclusive Read, Shared Write (ERSW) |
| 7% | Exclusive Read, Exclusive Write (ERW) |

If the file is opened without specifying the MODE, the default access mode is Byte I/O and SREW.

Refer to the 8800 Series Monroe Operating System Programmer's Reference Manual for additional information.

Examples:

Ex. 1

To open a file for Record I/O and ERW:

```
10  OPEN "VOL:FILE" AS FILE 1 MODE 7%¶
20  OPEN AS AS FILE 1 MODE 7%¶
```

Ex. 2

To open a file for Byte I/O and ERW:

```
10  OPEN "VOL:FILE" AS FILE 1 MODE 192%+7%¶
```

Ex. 3

To open a file for Byte I/O and SREW:

```
10  OPEN "VOL:FILE" AS FILE 1¶
```

Ex. 4

To open a file with type Bin for record I/O and exclusive write only:

```
10  OPEN "VOL:FILE/B" AS FILE 1 MODE 3%¶
```

## OUT Statement

**Function:** Sends data to the out-ports specified.

**Mode:** Direct/Program

**Format:** OUT <port,data> [,port,data...]

**Arguments:** If numeric constants are specified, they will be evaluated as decimal integers. The different ports available are listed in Appendix D.

**Use:** This is a machine-oriented statement meant for advanced programming.

The statement, used in conjunction with the INP function gives the user access to the I/O-handling of the system.

**Note:** The user should be familiar with I/O device programming. This statement may cause the system to crash. Refer to Z80-SIO-CTC and PIO Technical Manuals.

**Example:**

```
LIST¶
10  !
20  ! SIMPLE EXAMPLE SHOWING HOW TO SET UP NUMBER OF
25  ! STOP BITS IN SIO CHANNEL A
30  !
35  ! SELECT REGISTER 4 IN SIO. IN THIS EXAMPLE HAS
36  ! SIO COMMAND PORT 165.
40  OUT 10%*16%+5%,4%
50  ! SELECT 2 STOP BITS and X16 CLOCK.
55  OUT 10%*16%+5%,64%+8%+4%
60  !
70  ! CAUTION !! THIS ROUTINE WILL ONLY WORK UNDER THE
75  ! CONDITIONS THAT NO OTHER TASK OR INTERRUPT ROUTINE
```

```
 80  ! CHANGES THE SELECT REG 4 BETWEEN LINES 40 AND 50.
 85  ! E.G. THERE COULD BE OTHER INTERRUPT ROUTINES WHICH
 90  ! ARE ENABLED IN THE SYSTEM USING THE SIO DURING
 95  ! EXECUTING OF LINE 40 AND 50.
110  END
BASIC
```

## PEEK Statement

**Function:** Returns the memory contents (of 1 byte) of a specified address.

**Mode:** Direct/Program

**Format:** PEEK(<address>)

**Argument:** Address is the byte in memory to be accessed. It is specified in decimal.

**Result:** Integer

**Use:** PEEK is mainly used when Monroe BASIC works together with Assembler subroutines.

**Example:**

```
;SYS(11) ! FIND START OF USER PROGRAM AREA¶
29999
BASIC
10  REM PEEK:----RETURNS THE CONTENTS OF THE GIVEN¶
15  REM ARGUMENT (ADDRESS)¶
30  REM POKE:----IS USED TO INSERT A DATA(VALUE) INTO¶
35  REM A SPECIFIED LOCATION IN USER'S PROGRAM AREA¶
40  REM USE CAUTION WHILE USING THIS STATEMENT, IT MAY¶
45  REM CRASH THE SYSTEM OR DESTROY THE PREVIOUS¶
50  REM CONTENTS¶
80  A%=29999 ! ASSIGN MEMORY LOCATION IN USER PROGRAM AREA¶
90  Z%=PEEK(A%)¶
100 PRINT "----------" Z%¶
110 POKE A%,60¶
120 POKE A+Z,8¶
130 PRINT "----------A%=" A%¶
140 PRINT "----------A+Z=" A+Z¶
150 PRINT "----------" PEEK(A)¶
160 PRINT "----------" PEEK(A+Z)¶
170 END¶
RUN¶
----------60
----------A%=29999
----------A+Z=30059
----------60
----------8
```

## PEEK2 Function

**Function:** Reads the contents of two bytes.

**Mode:** Program/Direct

**Format:** PEEK2(<address>)

**Argument:** Address is the starting byte in memory to be accessed.

**Result:** Integer

**Use:** PEEK2 is mainly used when Monroe BASIC works together with Assembler subroutines.

**Example:**

```
10 A% = PEEK2(1234%)¶
20 ;A%¶
RUN¶
-3763
BASIC
```

**Note:** The above example is for illustration purposes only. The result will vary depending on the memory contents of locations 1234 and 1235.

## POKE Statement

**Function:** Changes or loads a specific value into a designated address in the user's program area in RAM.

**Mode:** Direct/Program

**Format:** POKE <address>,<data>[,data,...]

**Arguments:** Address is the starting byte in memory where the data is to be loaded. It is specified in decimal.

Data is the decimal equivalent of the 8-bit binary number to be set.

If more than one DATA-value is given the address is incremented one step for each new data value.

**Use:** Poke is mainly used when Monroe BASIC works together with assembler subroutines.

The address of the start of the user's program area can be found by the SYS(11) function. If a protected area of memory is POKEd the following message will be displayed on the console:

```
xx.yy.zz  WRITE prot at nnnnn
xx.yy.zz  PAUSED
-
```

The dash indicates you are back at the operating system level. Enter CO¶ and you will be back in BASIC.

**Caustion:** If POKE is used erroneously it may destroy the contents of needed memory locations.

**Example:**

```
;SYS(11) ! FIND START OF USER'S PROGRAM AREA¶
-287
BASIC
10 ; PEEK (-287)¶
20 POKE -287,0¶
30 ; PEEK (-287)¶
RUN¶
184
0
BASIC
```

## POSIT Statement

**Function:** Positions the file pointer to record or byte position desired or returns the current position of the pointer.

**Mode:** Program/direct

**Format:**
1. POSIT #<channel no.>,<position>
2. POSIT (<channel no.>)

**Arguments:** Channel no. corresponds to the internal channel number on which the file is opened.

Position is either the number of records or the number of bytes from the beginning of the file where access is to begin. Position "0" is the first record or first byte. Record or byte access is determined by the MODE specification in the previous PREPARE or OPEN statement. Refer to these statement discussions in this section for details.

The position supplied or returned is a floating point number.

**Use:** Each data file contains a pointer specifying the present position in records or bytes from the beginning of the file. This pointer can be read or positioned to a specific byte position using POSIT.

Format 1, above, is used to move the file pointer the specified number of records (or bytes) from the beginning of the file (the first position). The first position = 0. POSIT can be used together with all file handling instructions.

Format 2, above, yields the current position of the file pointer.

Examples:

Ex. 1 -

Read-Record I/O, Random starting point and subsequent (Sequential Read).

```
LIST¶
10 OPEN "VOL:DATA" AS FILE 1 MODE 7%
20 INPUT "STARTING RECORD NO.?" T%
30 INPUT "NUMBER OF RECORDS TO BE READ?" N%
40   POSIT #1, T%-1%
50 FOR I%=T% TO N% + T%
60   GET #1,AS COUNT S% ! S% IS THE RECORD LENGTH
70 NEXT I%
BASIC
```

Ex. 2

Read-Record I/O (Random Access)

```
LIST¶
10 OPEN "VOL:DATA" AS FILE 1 MODE 7%
20 POSIT #1,I% ! I% IS THE RECORD NUMBER
30 GET #1,A$ COUNT S%
BASIC
```

## PREPARE Statement

**Function:** Creates and opens a new file for sequential or random access on a file-structured device (diskette) with an I/O channel number internal to the Monroe BASIC program. Any existing file with the same name is deleted.

**Mode:** Direct/Program

**Format:** PREPARE <string expression> AS FILE <channel no.> [MODE a%+b%]

**Arguments:** String Expression specifies the name of the disk file to be opened.

The Channel no. after AS FILE must have an integer value corresponding to the internal channel number on which the field is opened. Numbers 1 through 250 are legal.

a% and b% are integers which determine mode I/O and Read/Write characteristics, as follows:

| a% | Meaning |
|---|---|
| 0% | Record I/O is to be used. Record length must be specified in GET statement. |
| 128% | Physical I/O on disk sector level. If 128 is specified data in excess of 256 bytes will be written in blocks of 256 bytes. The first record will consist of 256 bytes, the second 256 bytes, etc. H ence the fastest way to write is in blocks of 256 bytes (i.e., 256, 512, 768, etc.) |
| 192% | Byte I/O is to be used (default) |

| b% | Meaning |
|---|---|
| 0% | Sharable Read Only (SRO) |
| 1% | Exclusive Read Only (ERO) |
| 2% | Shared Write Only (SWO) |
| 3% | Exclusive Write Only (EWO)<br>Normally when a file is opened Read and Writes take place from beginning of a file. However, this special mode will start at the end of file (append) if no positioning is done. |
| 4% | Shared Read/Write (SRW) |
| 5% | Sharable Read, Exclusive Write (SREW) (default) |
| 6% | Exclusive Read, Shared Write (ERSW) |
| 7% | Exclusive Read, Exclusive Write (ERW) |

If the file is opened without specifying the MODE, the default access mode is Byte I/O and SREW.

Refer to the 8800 Series Monroe Operating System Programmer's Reference Manual for additional information.

Note: For allocation of files with fixed record length refer to example under SVC statement in this section.

Examples:

Ex. 1

To create and open a new ASCII file for Byte I/O and ERW:

```
10  PREPARE "VOL:FILE" AS FILE 1 MODE 192%+7%
```

Ex. 2

To create and open a new binary file for Byte I/O and ERW:

```
10  PREPARE "VOL:FILE/B" AS FILE 1 MODE 192% + 7%
```

SVC Statement

Function: Communicates with the operating system to perform special functions.

Mode: Program

Format: SVC <x%>,<A%>[,b%][,d%]

Arguments: x% is the Supervisor Call Number desired, as follows:

| x | Function |
|---|---|
| 1 | General Purpose I/O Requests |
| 2 | Memory Handling (2.1) |
| | Log Message (2.2) |
| | Pack File Descriptor (2.3) |
| | Pack Numeric Data (2.4) |
| | Unpack Binary Number (2.5) |
| | Fetch/Set Date/Time (2.7) |
| | Scan Mnemonic Table (2.8) |
| | Open/Close Device (2.12) |
| 3 | Timer Requests |
| 4 | Task Device |
| 5 | Loader Handling |
| 6 | Task Request |
| 7 | File Request |
| 8 | Resource Handling |

Each of the above is discussed in detail in the Monroe Operating System Reference Manual.

A% is an integer array parameter (SVC Block) specifying the parameters required by the SVC. The number of elements is dependent on the SVC which is called. Refer to the Monroe Operating System Reference Manual for contents and size of SVC block.

b% and d% are used with SVC 6 and represent switch settings. SVC6 allows Task Programs to be called by a Monroe BASIC program. These switches may be used to inform the task program how to act in different situations. Switches b% and d% are shown as alphabetical letters in the command formats in the Monroe Utility Programs Programmer's Reference Manual. They correspond to b% and d% as follows:

| b% - | 0 through Z |
|---|---|
| 1% | O |
| 2% | R |
| 4% | S |
| 8% | T |
| 16% | U |
| 2%**5% | V |
| 2%**6% | W |
| 2%**7% | X |
| 2%**8% | Y |
| 2%**9% | Z |

| d% - | A through P |
|---|---|
| 1% | A |
| 2% | B |
| 4% | C |
| 8% | D |
| 16% | E |
| 2%**5% | F |
| 2%**6% | G |
| 2%**7% | H |
| 2%**8% | I |
| . | . |
| . | . |
| . | . |
| 2%**15% | P |

Suppose, for example, the switch settings were V and 0. b% and d% would be represented as 2%**14% and 2%**5% respectively:

```
SVC 6%,A%,2%**5%,2%**14%
```

Use: This function is used when the programmer finds that a particular task cannot be performed with the available Monroe BASIC commands. SVC can accomplish the desired task by passing parameters specified by the user. The SVC routine uses these parameters to achieve desired results.

Caution: If this statement is used incorrectly, it may cause the system to crash.

Examples:

Ex. 1

```
LIST¶
10  ! THIS IS AN EXAMPLE HOW TO ALLOCATE
20  ! A FILE WITH FIXED RECORD LENGTH
30  ! USING SVC 7.
40  !
50  DIM A%(8%) ! DIMENSION SVC BLOCK.
60  ! ASSIGN A VARIABLE A FILE NAME.
70  ! FILE NAME MUST HAVE UNPACKED FORMAT.
80  ! THIS FILE WILL BE ALLOCATED ON SYSTEM VOLUME.
90  A$="    FILE"+SPACE$(20%)
100 A%(0%)=1% ! ALLOCATE.
110 A%(1%)=16%*256%+1% ! ASCII FILE AND LU=1.
120 A%(2%)=VARPTR(A$) ! POINT TO FILE NAME.
130 A%(4%)=32% ! RECORD LENGTH.
135 ON ERROR GOTO 145
140 SVC 7%,A%
142 GOTO 150
145 ; "ERROR = ":; ERRCODE
150 END
BASIC
```

Ex. 2

```
LIST¶
10  !
20  ! THIS EXAMPLE SHOWS HOW TO READ FROM CONSOLE
30  ! WHEN USER HAS A LIMITED FIELD SIZE.
40  ! A CARRIAGE RETURN, FUNCTION, OR CURSOR KEY
45  ! TERMINATES THE INPUT AND CURSOR EDITING
50  ! FACILITIES ARE SUPPORTED.
70  ! THIS IS VERY USEFUL IN DATA ENTRY PROGRAMS.
80  !
90  INTEGER : EXTEND
100 DIM SVCBLK(4) ! CREATE SVC BLOCK.
110 OPEN "CON:" AS FILE 1
120 A$=FNKEYIN$(23,20,5)
125 IF A$='' THEN 260 ! ''=TWO APOSTROPIES
130 ; "****OK" : GOTO 120
140 !
150 DEF FNKEYIN$(YPOS,XPOS,MAX) LOCAL IN$=80
160 PRINT CUR(YPOS,XPOS) SPACE$(MAX);CUR(YPOS,XPOS);
170 IN$=SPACE$(MAX)
180 SVCBLK(0)=17 ! RANDOM TO AVOID CR, LF AND IMAGE
185 ! ASCII.
190 SVCBLK(1)=1 ! LU
200 SVCBLK(2)=VARPTR(IN$) ! BUFFER ADDRESS.
210 SVCBLK(3)=MAX ! BUFFER SIZE.
220 SVC 1,SVCBLK
230 TERMINATOR=SWAP%(SVCBLK(1)) AND 255 ! GET TS.
240 RETURN LEFT$(IN$,SVCBLK(4))
250 FNEND
260 END
BASIC
```

Additional examples of SVC's can be found in the examples in Appendix C.

## SWAP Function

**Function:** Returns an integer with the first and second bytes transposed.

**Mode:** Program.

**Format:** SWAP%(n%)

**Arguments:** n% is an integer.

**Result:** Integer

**Example:**

```
10 A%=512% ! ASSIGN AN INTEGER A VALUE.¶
20 !¶
30 ! THE CORRESPONDING BIT CONFIGURATION OF A% IS¶
35 ! 00000010 00000000 (binary) = 512 (decimal)¶
40 ! CONTAINING 16 BITS OR SAME AS 2 BYTES¶
45 ! SWAP% FUNCTION SWAPS¶
50 ! THESE TWO BYTES SO RESULT WILL BE¶
55 ! 00000000 00000010 => (binary) = 2. (decimal)¶
60 !¶
70 B%=SWAP%(A%)¶
80 ; B%¶
90 END¶
 RUN¶
 2
BASIC
```

SYS Function

Function: Returns essential system information.

Mode: DIRECT/PROGRAM

Format: SYS(i%)

Argument: i can have the values shown below:

SYS(0) - Reserved for future use, and will presently cause an error 143 (ILLEGAL SYS FUNCTION) if used.

SYS(1) - Is reserved for future use, and will presently cause error 143 if used.

SYS(2) - Returns total space available for program.

SYS(3) - Returns current program size.

SYS(4) - Returns space left in user's program area.

SYS(5) - Returns the key operate flag, cleared by GET or INPUT; 0 if no key has been typed, 128 if there is a character available.

SYS(6) - Restores the last input character into the keyboard buffer.

SYS(7) - Returns the ASCII value of the key that was used to terminate the last user input to an INPUT or INPUT LINE statement. For a complete list of the terminating keys and their ASCII values refer to Table 2-1.

SYS(8) - Is reserved for future use, and will presently cause an error 143 if used.

SYS(9) - Reserved for future use, and will presently cause an error 143 if used.

SYS(10) - Returns a pointer (address) to information block about the program.

SYS(11) - Returns starting address of user program area.

SYS(12) - Returns a pointer (address) to the variable root for all variables in Monroe BASIC.

Result: Integer

## VAROOT/VARPTR Statement

**Function:** VAROOT returns the starting address of a table (or root) which contains information about a variable.

VARPTR returns the starting address of where a variable is stored.

**Mode:** Program/Direct.

**Format:** VAROOT(<variable>)
VARPTR(<variable>)

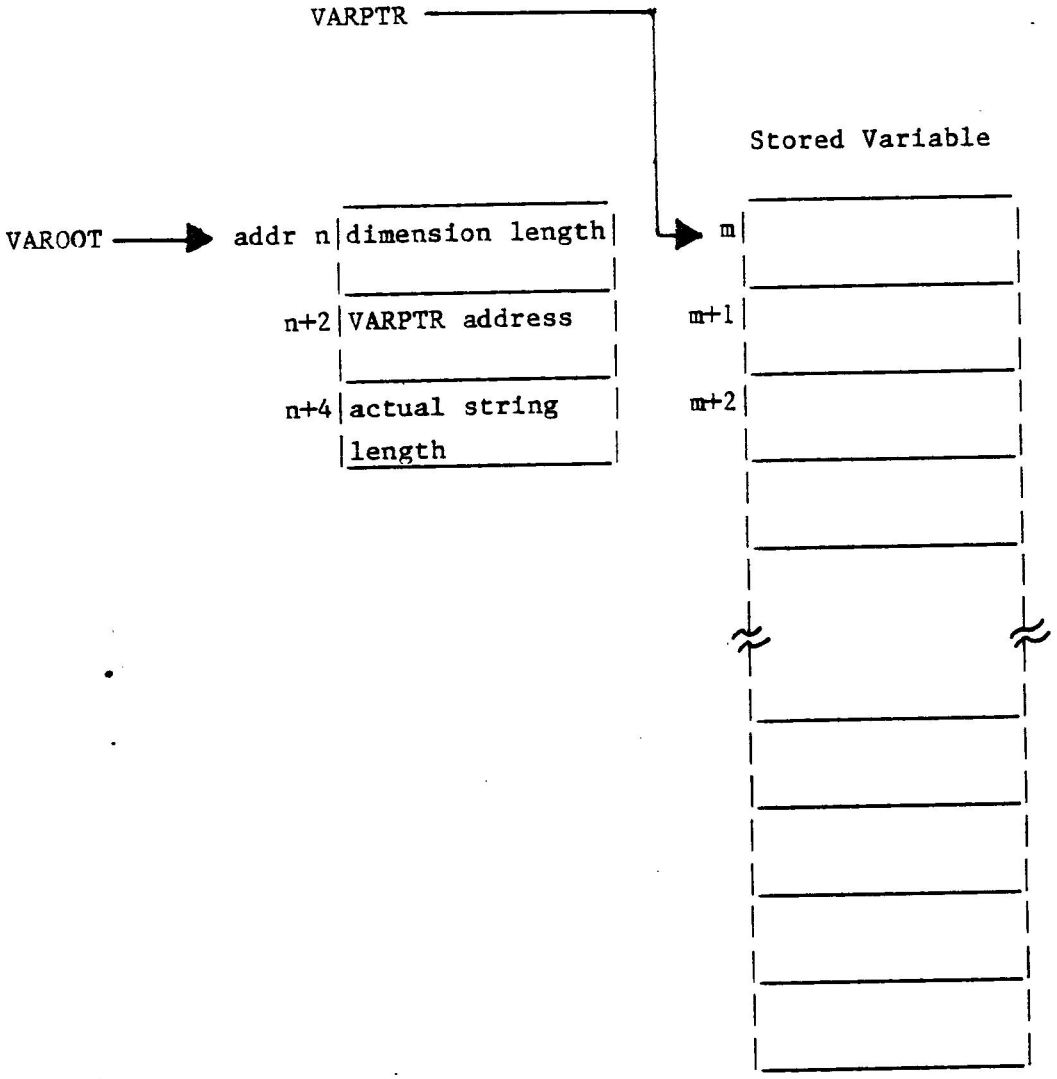
**Arguments:** Variable can be a string variable or arrays of any kind (integer, string, or floating point).

**Result:** Integer

**Use:** VARPTR can be used with the SVC statement to set-up buffer pointers, for example to SVC1 or 7, to where to PUT or GET desired data. It can also be used to locate the address of a variable. The variable can then be seen via PEEK and changed, if desired, via POKE.

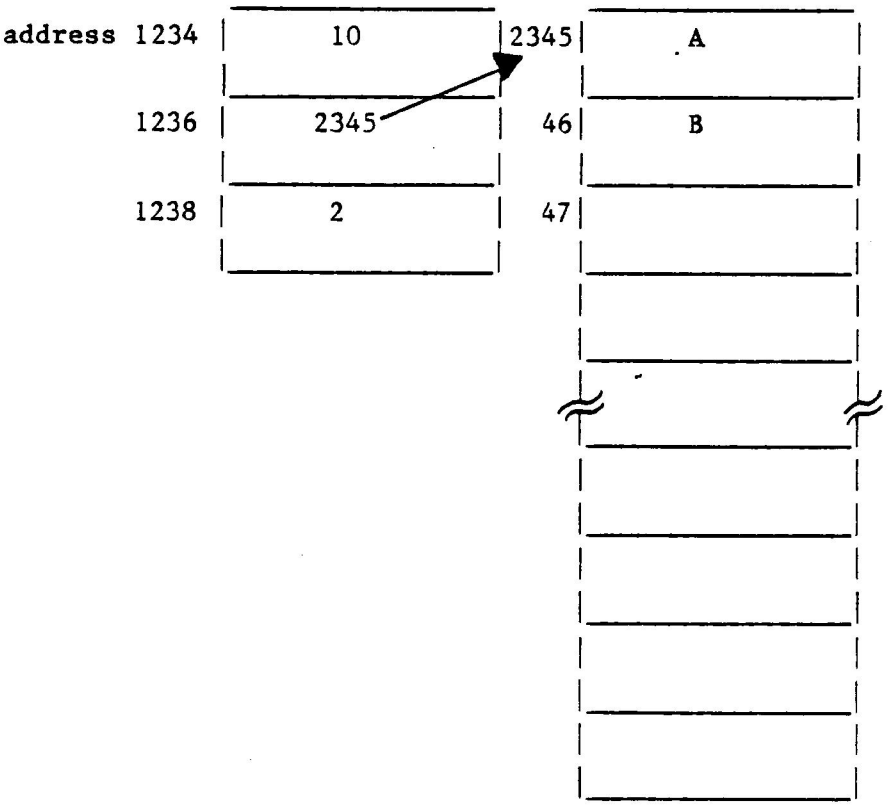
VAROOT is used in conjunction with the SVC statement to change the actual length of a string variable. For string arrays. It can be used to find the address of a root table containing the addresses of where individual array elements are stored.

Examples: 1. Use of VARPTR and VAROOT for string variables



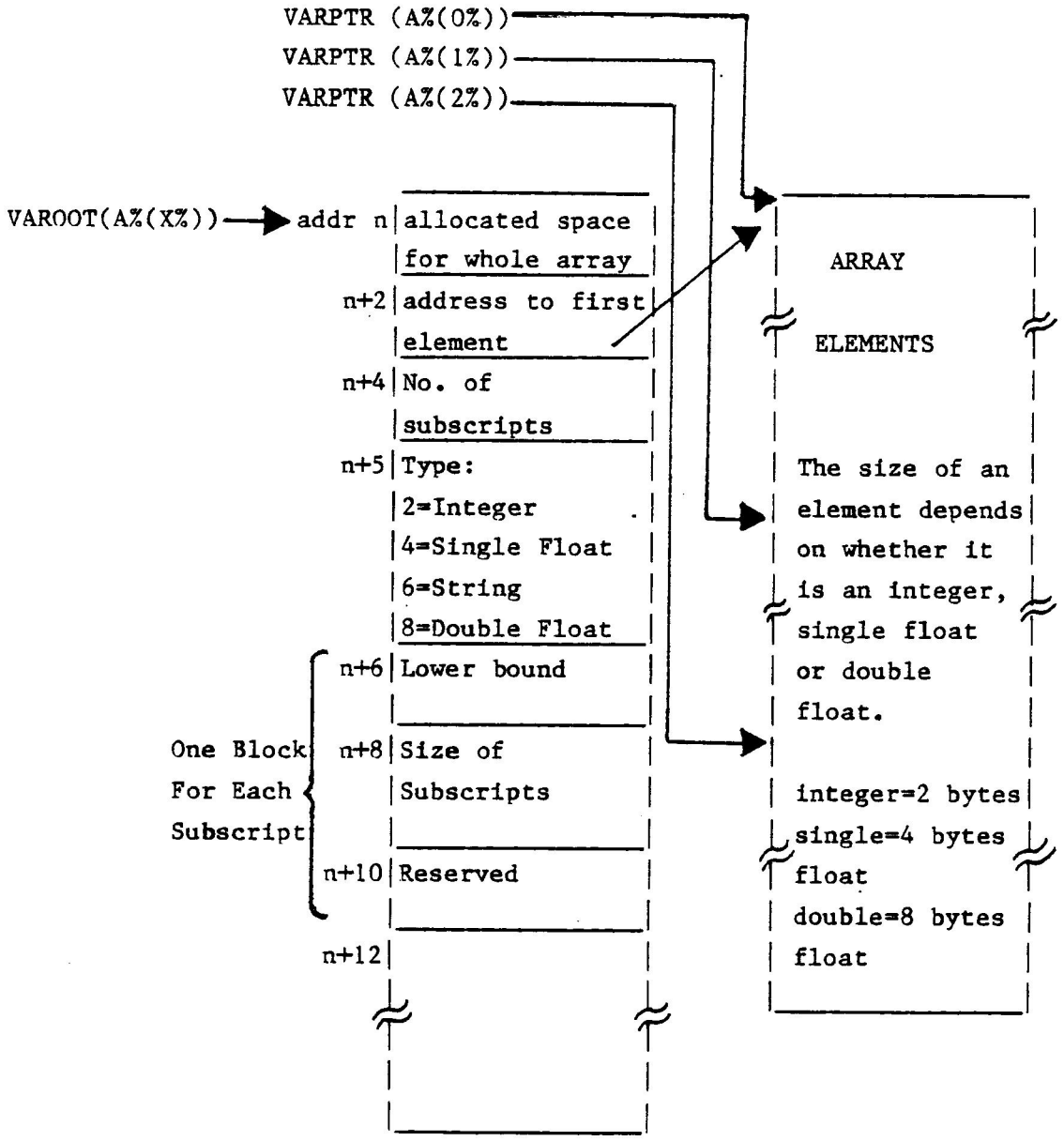
```
10  DIM X$=10¶
20  X$="AB"¶
30  ;VARPTR (X$)¶
40  ;VAROOT (X$)¶
50  END¶
RUN¶
2345
1234
BASIC
```

The above example shows the use of VARPTR and VAROOT for string variables. Here VARPTR points to location 2345 where the string AB is stored.




NOTE: This example is for illustration purposes only. Program lines 30 and 40 can give various results depending on where the variable resides.

Examples: 2. Use of VAROOT and VARPTR for arrays (integer, single, double float)

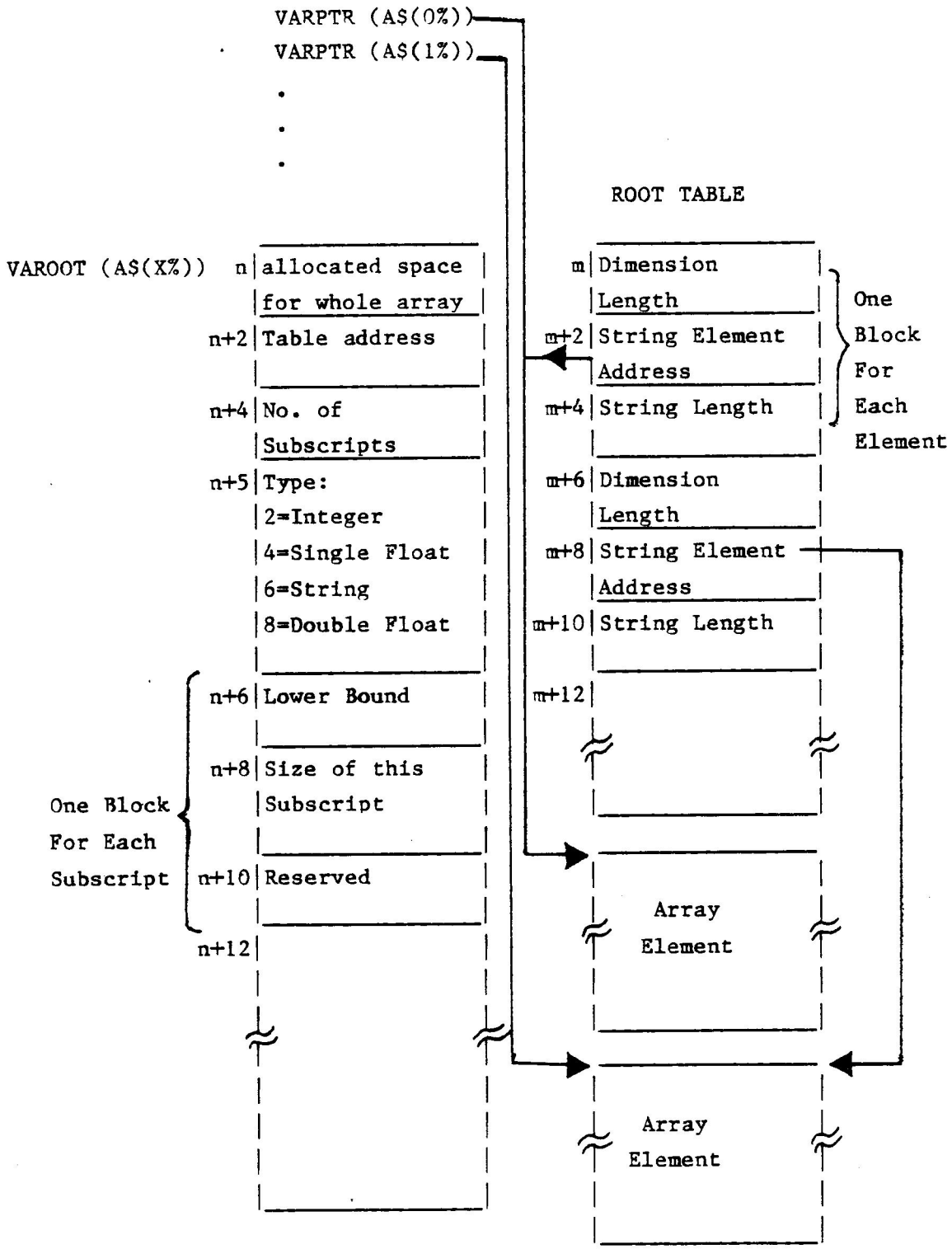



NOTE: VAROOT always points to the variable table independent of the index of the variable.

```
10  DIM A%(3%) ! ALLOCATED INTEGER ARRAY.¶
20  A%(1%)=10%¶
25  ! POINTER TO VARIABLE A%(0%)¶
30  ; VARPTR (A%(0%))¶
35  ! POINTER TO VARIABLE A%(1%)¶
40  ; VARPTR (A%(1%))¶
45  ! POINTER TO VARIABLE ROOT FOR WHOLE ARRAY.¶
50  ; VAROOT(A%(0%))¶
55  ! POINT AT SAME AS ABOVE.¶
60  ; VAROOT(A%(1%))¶
65  ! PRINT CONTENTS OF VARIABLE A%(1%)¶
70  ; PEEK2(VARPTR(A%(1%)))¶
75  ! PRINT TYPE AND NUMBER OF INDEX IN ARRAY.¶
80  ; PEEK2(VAROOT(A%(1%))+4%)¶
90  END¶
RUN¶
-5918
-5916
-5930
-5930
10
513
BASIC
```

NOTE: This example is for illustration purposes only. Program lines 30, 40, 50, and 60 can give various results depending on where the variable resides.

3. Use of VAROOT and VARPTR for string arrays.



```
5   ! ALLOCATE A STRING ARRAY¶
10  DIM A$(3%)=10%¶
20  A$(1%)="A"¶
25  ! POINTER TO VARIABLE A$(0%)¶
30  ; VARPTR (A$(0%))¶
35  ! POINTER TO VARIABLE A$(1%)¶
40  ; VARPTR (A$(1%))¶
45  ! POINTER TO VARIABLE ROOT FOR WHOLE ARRAY¶
50  ; VAROOT (A$(0%))¶
55  ! POINT AT SAME AS ABOVE¶
60  ; VAROOT(A$(1%))¶
65  ! PRINT ASCII VALUE OF FIRST CHAR. IN A$(1%))¶
70  ; PEEK2(VARPTR(A$(1%)))¶
75  ! PRINT ACTUAL LENGTH OF A$(1%)¶
80  ; PEEK2(PEEK2(VAROOT(A$(1%))+2%)+10%)¶
90  END¶
RUN¶
-5894
-5884
-5930
-5930
65
1
BASIC
```

NOTES: This program is for illustration purposes only. Program lines 30, 40, 50, and 60 can give various results depending on where the variable reside.

VAROOT always points to the variable table independent of the index of the variable.

## 14.3 FILE CREATION

Monroe BASIC supports data files with the following types of records:

1. Variable Length Records
2. Fixed Length Records

That is, the size of the file subdivisions to which records correspond may be either uniform (fixed) length or variable length. The choice of record type is determined by the anticipated use of the file.

### Variable Length Records

To allocate a data file of variable length records use the PREPARE statement. Data can be loaded, for example, using a PUT or PRINT statement as shown in the procedure below.

Examples:

```
PREPARE "FILEA" AS FILE 1
INPUT "NUMBER OF RECORDS?" R%
  FOR I% = I% to R%
    INPUT "ASCII DATA?" AS
    PRINT #1%,A$
  NEXT I%
```

```
PREPARE "FILA/B" AS FILE 2
! FILEA/B SPECIFIES BINARY
DATA FILE
INPUT "NUMBER OF RECORDS?" R%
FOR I%=I% TO R%
INPUT "BINARY DATA?" A$
PUT #2%, A$
NEXT I%
```

### Fixed Length Records

To allocate a file with fixed length records requires the execution of a particular Monroe BASIC program. This program should request the operating system to perform this function via supervisor calls. These calls are discussed in detail in the Monroe Operating System Programmer's Reference Manual. Monroe BASIC programs that accomplish this function are shown in Appendix C. Program FIXLEN prompts the user for the file name and the record length and then creates the file.

An ISAM index file and its associated data file are created by Utility Program CREINDEX. The execution of this program initiates interactive prompting for key and data information. After this information is entered the ISAM index and data files are allocated.

## 14.4 ACCESS METHODS

The method of accessing a data file is determined by whether the file contains:

1. Variable Length Records
2. Fixed Length Records

### Variable Length Records

Data files containing variable length records are accessed sequentially, with or without a random starting point, as shown in the following procedure:

1. Specify OPEN statement with Byte I/O and READ/WRITE mode desired.

   Examples:

```
OPEN "VOL:FILELIST/B" AS FILE 1 !BYTE I/O (DEFAULT)
OPEN "VOL:FILEWRITE" AS FILE 2 MODE 192% + 1%
```

2a. Sequential Access - Use the GET statement in conjunction with a loop, for example, to specify the number of bytes you want to access for N number of times.

   Example:

```
INPUT "ENTER N?" N%
INPUT "NUMBER OF BYTES TO BE READ?" S%
  FOR I% = 1% to N%
  GET #1, A$ COUNT S%   !S% IS NUMBER OF BYTES
  ! IF COUNT IS OMITTED, ONE BYTE WILL BE READ
.
.
.
NEXT I%
.
.
.
```

2b. Sequentially with Random Starting Point - Use the POSIT statement to select random starting point in the data file where access is to begin. The GET statement can then be used to access a specific number of bytes. Loops can also be constructed, if desired, as in 2a above.

Example:

```
POSIT #1,99 !POSITIONS POINTER TO 100TH BYTE
GET #1,A$ COUNT 10
PRINT A$ !PRINTS THE 100TH TO 109TH BYTES IN FILE
```

Fixed Length Records

Data files containing fixed length records can be accessed Sequentially or Randomly. The great advantage of working with fixed-length records is that they allow you to place data directly into or extract it directly from any file location you specify.

Sequential Access: The procedure specified previously for variable length records also applies for Sequential access of fixed length record files. Refer to this section for details.

Random Access: To randomly access a file containing fixed length records follow the procedure shown below:

1. Specify OPEN statement with Record I/O and Read/Write mode desired.

   Examples:

   ```
   OPEN "VOL:FIXED/B" AS FILE 1 MODE 0%
   OPEN "VOL:SSNUM/B" AS FILE 2 MODE 0%+3%
   ```

2. Specify the POSIT statement to select random point in the data file where access is to occur.

   Example:

   ```
   POSIT #1, 99 !POSITIONS POINTER TO 100TH RECORD
   ```

3. Specify the GET statement to access the record where the pointer is located. Loops can also be constructed to access a specific number of records as shown below.

Example:

```
!FOR SINGLE RECORD ACCESS
!RECORD LENGTH (e.g., 28) MUST BE SPECIFIED
GET #1, A$ COUNT 28
PRINT A$  !PRINTS THE 100TH RECORD

or

! *** FOR MULTIPLE RECORD ACCESS ***
!
INPUT "NUMBER OF RECORDS TO BE ACCESSED?" N%
INPUT "STARTING RECORD NUMBER?" I%
POSIT #2,I%-1%
FOR I% = 1% TO N%
  GET #2, A$ COUNT 28
  .
  .
  .
  NEXT I%
.
.
.
```

# SECTION 15
# LOW RESOLUTION
# BUSINESS GRAPHICS

# SECTION 15
# LOW RESOLUTION BUSINESS GRAPHICS

## 15.1 INTRODUCTION

Low resolution business graphics is available on Monroe's 8800 Series Occupational Computer. It is implemented as part of Monroe BASIC which allows information to be displayed on the console in various graphics modes via the PRINT (;) command. These modes are enabled by specifying particular decimal values in the CHR$ function as part of a PRINT statement. The user has the option of using these modes either individually or in combination, depending upon the function and desired format of the output. The console output can appear with such attributes as: blinking, half intensity, double height, double width, etc. Special rectangular graphic blocks can also be displayed separately or in combination to form desired shapes, letters, borders, etc.

## 15.2 GRAPHICS CHARACTERS

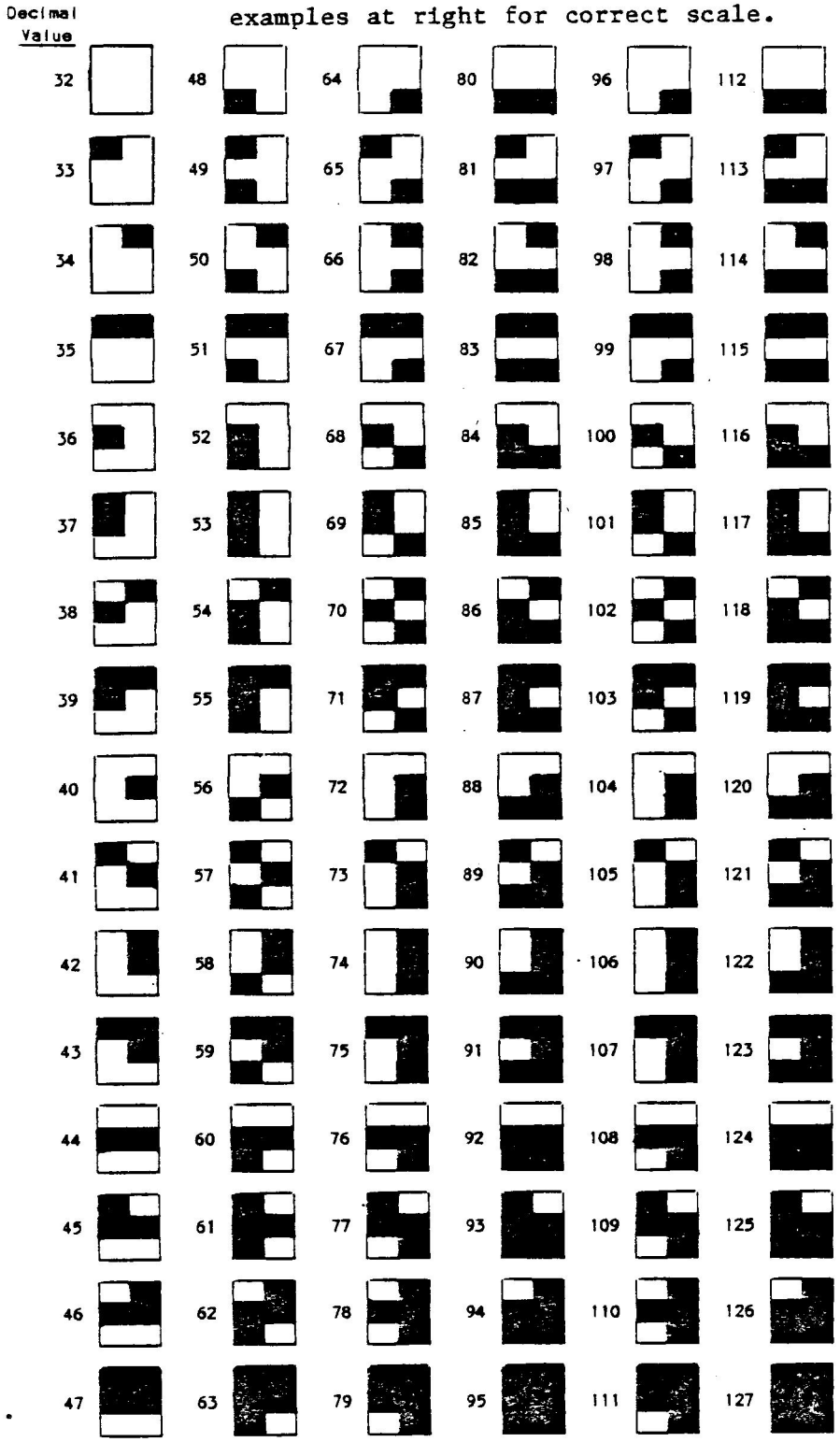
Printable characters are characters which have ASCII values of 32 to 127. In a PRINT statement these characters can be enclosed in quotes (e.g., "TEST"), can be represented by variables (e.g., A$ or B), or can be specified in functions (e.g. CHR$(33)). This set displays:

1. text characters-
   a. letter A through Z and a through z
   b. numbers 0 through 9
   c. special printer characters and symbols

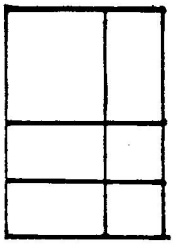
2. block graphics characters - any of the block graphics characters shown in Figure 15-1. Note that these block characters are not drawn to scale.

Unprintable characters having decimal values of 128 or greater are treated as graphics mode attributes and are called graphics control characters. They enable the display of text characters or graphic blocks with particular appearance attributes. These control characters are not displayed on the screen. Whether a character is displayed as a text or graphics block character is determined by the graphics control character (mode) previously specified for that line.

Note: Boxes below are not drawn to scale. See examples at right for correct scale.




Drawn to scale (8x)



118

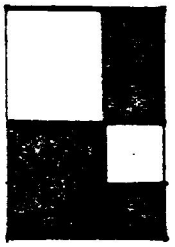


Figure 15-1. Block Graphics Character Images

## 15.3 GRAPHICS MODES

The following business graphics modes are available:

| Mode | Decimal Value | Meaning |
|---|---|---|
| Normal Text/ Reset | 128 | Sets the type of text to normal Graphics text. It can also be used to disable other modes in effect. |
| Double Upper Half | 130 | Sets the type of text to Graphics text and displays the upper half of each character in double height. Used in conjunction with "129". |
| Double Lower Half | 129 | Sets the type of text to Graphics text and displays the lower half of each character in double height. Used in conjunction with "130". |
| Block Graphics | 139 | Sets the type of text to Graphics text and the type of characters to block graphics characters. These are shown in Figure 15-1. |

## 15.4 GRAPHICS ATTRIBUTES

Once a graphics mode is selected there are a variety of different format types available for the graphic output. These are accomplished by "oring" the indicated mode with one or more of the decimal values (i.e., CHR$(mode % + attribute% + attribute% ...) as shown below.

| Attribute | Decimal Value | Meaning |
|---|---|---|
| Dim | 64 | Half illuminates each character. |
| Elongate | 32 | Doubles width of alternate characters, ignores intervening alternate characters. |
| Reverse Video | 16 | Reverses background and text color. |
| Underline | 8 | Underlines each character. |
| Blink | 4 | Blinks each character. |

Each attribute can be used either singly or in combination with other attributes, for example:

CHR$(128 + 16) - Enables text to be displayed in reverse video.

CHR$(128 + 16 + 4) - Enables text to be displayed in reverse video and blinks the text.

CHR$(128 + 16 + 4 + 64) - Enables text to be displayed in reverse video, blinking and in half intensity.

CHR$(139 + 4) - Enables graphic blocks to be displayed each blinking on and off.

Note that if the "Blinking" and "reverse video" attributes are effected simultaneously (see Example 1 at the end of this section), the unlit (rather than the lit portions) of the character are alternately lit and unlit.

## 15.5 CONTROL CHARACTERS

ASCII characters having values less than 32 decimal are called control characters. The control characters shown below are acted upon in the indicated manner when sent to the console; other control characters are ignored. When these characters are output in a PRINT statement, a semicolon should normally follow the last item to prevent an unwanted carriage return/line feed from being sent to the screen.

| Name | Value | Action |
|---|---|---|
| Bell | 7 | Sounds the beeper. |
| Backspace | 8 | Moves the cursor one character to the left. |
| Tab | 9 | Moves the cursor one character to the right. |
| Line feed | 10 | Moves the cursor down one line. If the cursor is at the bottom of the screen, all the lines on the screen move up one line, the top line is lost, and the bottom line is cleared. Also cancels all graphic types in effect. |
| Form feed | 12 | Moves the cursor to the upper left hand corner of the screen and clears the screen. |
| Return | 13 | Moves the cursor to the beginning of the current line. |
| CTRL Q | 17 | Clears from the cursor to the end of the screen. |
| CTRL R | 18 | Clears from the cursor to the end of the line. |
| CTRL Z | 26 | Moves the cursor to the upper left hand corner of the screen and clears the screen. |

## 15.6 GRAPHICS PRINT FORMAT

The general format for displaying low resolution graphics on the console is:

PRINT [position] CHR$(<mode> + <attributes>) [CHR$(control)] <list>

where:

1. "Position" is optional and can be any one of the functions CUR(Row, Colunn) or TAB(Column).

2. "Mode" is any one of the modes given in the table in Section 15.3.

3. "Attributes" is any of the attributes given in the table in Section 15.4.

4. "List" can represent either a variable, expression, text image, or a function (such as CHR$, STRING$, or MID$). The user should carefully study the examples at the end of this section.

5. "Control" is any one of the control characters given in the table in Section 15.5.

## 15.7 ILLUSTRATED EXAMPLES

1. Display the character string "RECORDS" using the blinking and reverse video Graphics attributes.

```
;CHR$(128% + 4% + 16%) "RECORDS"
```

2. Display the character string "BASIC" using the dim and elongate attributes.

```
; CHR$(128% + 64% + 32%) "B A S I C "
```

3. Display a line of solid blocks ("127") across the console.

```
; CHR$(139) STRING$(80,127)
```

4. Display a line of blinking double horizontal bars ("83") starting at the 20th column position on the screen to the 40th column position.

```
; TAB(20) CHR$(139 + 4) STRING$(21,83)
```

Refer to the sample program in 14 below, for additional examples of using Functions with block graphics.

5. When multiple mode/attribute characters are included in a PRINT statement with no intervening line feed, CHR$(128) byte, or INPUT commands separating them, the attribute characters are "ored" together which in turn will effect the attributes of the output stream. For example, to display the string "ABCD" with "AB" underlined and "CD" both blinking and underlined, enter:

```
;CHR$(128% + 8%) "AB" CHR$(128% + 4%) "CD"
```

6. Display "AB" underlined and "CD" blinking only.

```
;CHR$(128% + 8%) "AB" CHR$(128% + 4%) "CD"
```

7. Display "AB" in reverse video and "CD" normal.

```
;CHR$(128% + 16%) "AB" CHR$(128%) "CD"
```

8. Display "AB" in dimmed normal form and "CDEFGH" in dimmed elongated (double width) form.

```
;CHR$(128% + 64%) "AB" CHR$ (128% + 32%) "C D E F G H "
```

(Note the required spaces between letters.)

9. Display "AB" in dimmed normal form, "CD" in dimmed elongated form, and "EFGH" in normal form.

```
;CHR$(128% + 64%) "AB" CHR$(128% + 32%) "C D " CHR$(128%)
"EFGH"
```

10. Contiguous output strings can be generated to fill an entire screen if necessary. Insertion of a CUR(Row, Column) to direct some of the output will not terminate the "oring" of attributes. If CUR(Row, Column) is omitted, the characters will wraparound to sequential lines.

11. The screen attributes of prompts and echoed responses can be controlled by the use of graphic modes/attributes as previously specified. However, the appearance on the screen of some combinations of attributes may not make sense or may not be suitable for this purpose.

12. The Double Upper Half Mode CHR$(130) and Double Lower Half Mode CHR$(129) are combined to double the height of a string of text.

```
;CHR$(130) "REPORT" CHR$(13,10,129) "REPORT"
```

13. The Elongate mode and the Double Upper and Double Lower Half modes can be combined to double the height and width of a string of text.

```
;CHR$(130 + 32) "R E P O R T " CHR$(13,10,129 + 32)
"R E P O R T "
```

14. To simplify the use of the Graphic Mode/Attributes you can equate them with variables and then use the variables in the CHR$ command and other functions. This is shown in the following two programs which illustrate the concepts discussed so far.

Program 1

```
10  ! ILLUSTRATION OF LOW RESOLUTION BUSINESS GRAPHICS MODES
20  ! THIS PROGRAM SUBSTITUTES ACRONYMS FOR THE VARIOUS GRAPHICS
30  ! MODES REPRESENTED BY CHARACTERS 128 - 139 AND THE TYPE
35  ! ATTRIBUTES REPRESENTED BY CHARACTERS 4-64.  THESE ARE
40  ! THEN USED TO PRINT A LINE OF TEXT.
50  EXTEND
60  INTEGER
70  Normal=128 ! NORMAL MODE
80  Dblkh=129 ! DOUBLE LOWER HALF MODE
90  Dbuh=130 ! DOUBLE UPPER HALF MODE
120  Gblck=139 ! BLOCK GRAPHICS MODE
140  !
150  ! ATTRIBUTES 'TYPES'
160  !
170  Dimi=64 ! DIM (HALF ILLUMINATION)
180  Elon=32 ! ELONGATE (DOUBLE WIDTH)
190  Rev=16 ! REVERSE VIDEO
200  Uline=8 ! UNDERLINE
210  Blink=4 ! BLINKING
250  !
260  ! EXAMPLE OF HOW TO USE THE ABOVE SUBSTITUTIONS
270  ! PRINT CHR$(MODE + ATTRIBUTE + ATTRIBUTE) "LITERAL TEXT"
280  !
290  Text$="THIS IS A LINE OF TEXT ILLUSTRATING THE ABOVE
291    SUBSTITUTIONS"
295  LARGETEXT$ = "S T A T U S   R E P O R T "
300  !
310  PRINT CHR$(normal+Dimi+Rev+Uline+Blink) Text$
320  PRINT
330  PRINT CHR$(Normal+Rev) Text$
340  PRINT
350  PRINT CHR$(Dbuh) Text$ CHR$(13,10,Dblkh) Text$
360  PRINT
370  PRINT CHR$(Dbuh + Elon) LARGETEXT$ CHR$(13,10,DBlkh +
       Elon) LARGETEXT$
380  END
```

Program 2

```
10  ! **** BUSINESS GRAPHIC BLOCK CHARACTERS ****
20  EXTEND
30  INTEGER
80  !
90  Block=139 ! BLOCK GRAPHICS
100  Blink=4 ! BLINKING GRAPHICS
320  ! GRAPHICS BLOCKS
330  ! SAMPLE BLOCKS - REFER TO FIG 15-1  FOR OTHERS
340  Ulcorn=55 ! UPPER LEFT CORNER BLOCK
350  Llcorn=117 ! LOWER LEFT CORNER BLOCK
360  Urcorn=107 ! UPPER RIGHT CORNER BLOCK
370  Lrcorn=122 ! LOWER RIGHT CORNER BLOCK
380  Upline=35 ! UPPER LINE BLOCK
390  Lline=112 ! LOWER LINE BLOCK
400  Mline=44 ! MIDDLE LINE BLOCK
410  Lbar=53 ! LEFT BAR BLOCK
420  Rbar=106 ! RIGHT BAR BLOCK
430  SBLCK=127 ! SOLID BLOCK
440  DHORZ=83 ! DOUBLE HORIZONTAL BARS
460  ! EXAMPLE
470  ! PRINT CHR$(Graphic Mode + type) "GRAPHIC BLOCKS"
480  !
485  PRINT "left bar" CHR$(Block) CHR$(Lbar)
487  PRINT
490  PRINT "left bar" CHR$(Block) STRING$(30,Lbar)
500  PRINT
510  PRINT 'upper line' CHR$(Block + Blink) STRING$(30, Upline)
520  PRINT
530  PRINT 'lower line' CHR$(Block) STRING$(30,Lline)
540  PRINT
542  PRINT 'solid block' CHR$(Block) CHR$(SBLCK)
543  PRINT
545  PRINT 'solid block' CHR$(Block+Blink) STRING$(30,SBLCK)
550  PRINT 'double horiz. bars' CHR$(Gwide) STRING$(30,DHORZ)
560  END
```

# APPENDIX A
# MONROE BASIC ASCII CHARACTER SET

# APPENDIX A
## MONROE BASIC ASCII CHARACTER SET

| Dec. | Char. | Dec. | Char. | Dec. | Char. | Dec. | Char. |
|---|---|---|---|---|---|---|---|
| 0 | NUL (CTRL [) | 32 | SPACE | 64 | @ | 96 | ` |
| 1 | SOH (CTRL a) | 33 | ! | 65 | A | 97 | a |
| 2 | STX (CTRL b) | 34 | " | 66 | B | 98 | b |
| 3 | ETX (CTRL c) | 35 | # | 67 | C | 99 | c |
| 4 | EOT (CTRL d) | 36 | $ | 68 | D | 100 | d |
| 5 | ENQ (CTRL e) | 37 | % | 69 | E | 101 | e |
| 6 | ACK (CTRL f) | 38 | & | 70 | F | 102 | f |
| 7 | BEL (CTRL g) | 39 | ´ | 71 | G | 103 | g |
| 8 | BS (CTRL h) | 40 | ( | 72 | H | 104 | h |
| 9 | TAB | 41 | ) | 73 | I | 105 | i |
| 10 | LF | 42 | * | 74 | J | 106 | j |
| 11 | VT (CTRL k) | 43 | + | 75 | K | 107 | k |
| 12 | FF (CTRL l) | 44 | , | 76 | L | 108 | l |
| 13 | CR | 45 | - | 77 | M | 109 | m |
| 14 | SO (CTRL n) | 46 | . | 78 | N | 110 | n |
| 15 | SI (CTRL o) | 47 | / | 79 | O | 111 | o |
| 16 | DLE (CTRL p) | 48 | Ø | 80 | P | 112 | p |
| 17 | DC1 (CTRL q) | 49 | 1 | 81 | Q | 113 | q |
| 18 | DC2 (CTRL r) | 50 | 2 | 82 | R | 114 | r |
| 19 | DC3 (CTRL s) | 51 | 3 | 83 | S | 115 | s |
| 20 | DC4 (CTRL t) | 52 | 4 | 84 | T | 116 | t |
| 21 | NAK (CTRL u) | 53 | 5 | 85 | U | 117 | u |
| 22 | SYN (CTRL v) | 54 | 6 | 86 | V | 118 | v |
| 23 | FTB (CTRL w) | 55 | 7 | 87 | W | 119 | w |
| 24 | CAN (CTRL x) | 56 | 8 | 88 | X | 120 | x |
| 25 | EM (CTRL y) | 57 | 9 | 89 | Y | 121 | y |
| 26 | SUB (CTRL z) | 58 | : | 90 | Z | 122 | z |
| 27 | ESC | 59 | ; | 91 | [ | 123 | { |
| 28 | FS(CTRL =) | 60 | < | 92 | \ | 124 | ¦ |
| 29 | GS (CTRL { ) | 61 | = | 93 | ] | 125 | } |
| 30 | RS (CTRL -) | 62 | > | 94 | ^ | 126 | ~ |
| 31 | US (CTRL ´) | 63 | ? | 95 | _ | 127 | DEL(CTRL `) |

## APPENDIX A - MONROE BASIC ASCII CHARACTER SET

| Dec. | Key | Dec. | Key | Dec. | Key | Dec. | Key |
|---|---|---|---|---|---|---|---|
| 128 | F1 | 160 | | 192 | PRNT SCRN | 224 | |
| 129 | F2 | 161 | | 193 | DEL CHAR | 225 | |
| 130 | F3 | 162 | | 194 | DEL LINE | 226 | |
| 131 | F4 | 163 | | 195 | | 227 | |
| 132 | F5 | 164 | | 196 | | 228 | |
| 133 | F6 | 165 | | 197 | | 229 | |
| 134 | F7 | 166 | | 198 | | 230 | |
| 135 | F8 | 167 | | 199 | HOME | 231 | |
| 136 | F9 | 168 | CTRL BS | 200 | CLEAR | 232 | |
| 137 | F10 | 169 | CTRL TAB | 201 | | 233 | |
| 138 | F11 | 170 | | 202 | INSRT LOCK | 234 | |
| 139 | F12 | 171 | | 203 | | 235 | |
| 140 | F13 | 172 | | 204 | | 236 | |
| 141 | F14 | 173 | CTRL CR | 205 | | 237 | |
| 142 | F15 | 174 | | 206 | | 238 | |
| 143 | F16 | 175 | | 207 | | 239 | |
| 144 | CTRL F1 | 176 | | 208 | RUN | 240 | |
| 145 | CTRL F2 | 177 | | 209 | LOAD | 241 | |
| 146 | CTRL F3 | 178 | | 210 | CONT | 242 | |
| 147 | CTRL F4 | 179 | CTRL ← | 211 | LEARN | 243 | |
| 148 | CTRL F5 | 180 | CTRL → | 212 | CALC | 244 | |
| 149 | CTRL F6 | 181 | CTRL ↑ | 213 | | 245 | |
| 150 | CTRL F7 | 182 | CTRL ↓ | 214 | | 246 | |
| 151 | CTRL F8 | 183 | | 215 | | 247 | |
| 152 | CTRL F9 | 184 | | 216 | | 248 | |
| 153 | CTRL F10 | 185 | | 217 | | 249 | |
| 154 | CTRL F11 | 186 | | 218 | | 250 | |
| 155 | CTRL F12 | 187 | | 219 | | 251 | |
| 156 | CTRL F13 | 188 | | 220 | | 252 | |
| 157 | CTRL F14 | 189 | | 221 | | 253 | |
| 158 | CTRL F15 | 190 | | 222 | | 254 | |
| 159 | CTRL F16 | 191 | | 223 | | 255 | STOP |

# APPENDIX B
# ERROR MESSAGES

# APPENDIX B
# ERROR MESSAGES

Table B-1 lists the error messages that can be returned when using Monroe BASIC. If an error is found during execution of a program the line number where the error occurred will be appended to the message. The file BASIC ERR/ASC must be on the system volume for the textual message to be printed; otherwise, just the error number will appear. Errors 10 to 80 below are associated with SVC's 1-8, respectively. Refer to the 8800 Series Monroe Operating System Programmer's Reference Manual.

Table B-1. Error Messages

| Number | Message | Meaning |
|---|---|---|
| 0 | Internal error. | |
| 1 | End of nodes. | Can't open any more files; seven already open. |
| 2 | Invalid function code. | Invalid SVC function code argument. |
| 3 | Can't connect to the resource. | |
| 4 | Resourece off line. | |
| 7 | Request is canceled. | |
| 10 | Illegal or unassigned LU. | |
| 11 | Access mode mismatch. | |
| 12 | Time out. | |
| 13 | Device off line. | |
| 14 | End of file. | |
| 15 | End of media. | |
| 16 | Recoverable or parity error. | |
| 17 | Unrecoverable error. | |
| 18 | Invalid random address. | |
| 19 | Nonexistent random address. | |
| 20 | Invalid subfunction number. | |
| 21 | Invalid file descriptor format. | |
| 23 | New volume already present. | |
| 24 | Directory device not closed. | |
| 30 | Invalid interval/time of day. | |
| 40 | Not assigned. | |
| 41 | Invalid device type. | |
| 50 | Illegal task name/number. | |

| Number | Message | Meaning |
|---|---|---|
| 51 | Task already present. | |
| 52 | Illegal priority. | |
| 53 | Illegal option. | |
| 54 | Illegal code/item at load. | |
| 55 | Overlay don't fit. | |
| 60 | Illegal task name/number. | |
| 61 | Task already present. | |
| 62 | Illegal priority. | |
| 63 | Illegal option. | |
| 64 | Event queue disabled. | |
| 65 | Invalid task status | |
| 66 | Invalid termination parameter. | |
| 67 | More items present in event queue. | |
| 70 | Assignment error. | |
| 71 | Illegal access mode. | |
| 72 | Size error. | |
| 73 | LU is not a direct access device. | |
| 74 | File descriptor of inv. format. | |
| 75 | Name error. | |
| 76 | Invalid key. | |
| 77 | File already exists. | |
| 80 | Illegal name/number. | |
| 81 | Illegal class. | |
| 82 | Already present. | |
| 83 | Parent not present. | |
| 84 | Dual not present. | |
| 85 | Invalid RCB-type | |
| 86 | End of memory. | |
| 120 | ISAM - key not found. | |
| 121 | ISAM - duplicate key. | |
| 122 | ISAM - illegal key value. | |
| 123 | ISAM - mismatch at check-read. | |
| 124 | ISAM - index not found. | |
| 125 | ISAM - bad data record length | |
| 126 | ISAM - task: end of memory. | |
| 130 | Floating point overflow. | Certain value out of range. |

| Number | Message | Meaning |
|---|---|---|
| 131 | Array Index outside of legal range. | Attempt to use an index greater than allowed for in the DIM. |
| 132 | Integer overflow | Integer out of range. |
| 133 | ASCII Arithmetic Overflow. | |
| 134 | String Index Neg. or too large. | |
| 135 | Negative TAB,SPACE$,STRING$ arg. | |
| 136 | Overflow in string assign. | The dimensions of the receiving string are too small. |
| 137 | Attempt to expand array or string. | A vector cannot be extended beyond its original length. |
| 138 | Expression of range in ON | |
| 139 | "RETURN" without GOSUB. | A return statement is encountered when no GOSUB has been executed. |
| 140 | Wrong return type. | |
| 141 | Out of DATA statements. | The data list is exhausted and a READ statement wants more data. |
| 142 | Wrong arg. to built in function. | |
| 143 | Illegal SYS function. | |
| 144 | Previously rejected line. | |
| 145 | DEF or FNEND entered the wrong way. | "DEF" or "FNEND" not preceded by RETURN |
| 146 | "PRINT USING" error. | Wrong format in PRINT USING statement. |
| 147 | Illegal data terminator. | |
| 148 | Insufficient data. | Too few data items typed at INPUT. |
| 149 | Restore to a non-data line. | "RESTORE" not on a "DATA" line. |
| 150 | Too much data. | Too many data items typed at INPUT. |
| 151 | Resume or ERRCODE without error. | |
| 176 | Dot address off screen. | |

| Number | Message | Meaning |
|---|---|---|
| 180 | Line not found. | Reference to a nonexistent line number. |
| 181 | Line is in wrong function def. | |
| 182 | Can't find matching NEXT. | |
| 183 | NEXT without matching FOR. | |
| 184 | Wrong variable after "NEXT". | |
| 185 | Nested FOR loops with same Var. | |
| 186 | FOR with local variable (Sorry). | Use of the FOR loop with a local variable is not permitted. This applies to multiple line function. |
| 187 | Undefined user function. | Call for undefined function. |
| 188 | Multiple defined user function. | More than one function with the same name. |
| 189 | Nested DEF:S illegal. | Mixing of several DEF instructions is not allowed. |
| 190 | Number of indices not consistent. | The number of indexes is not in accordance with the DIM statement. |
| 191 | Not assignable. | The argument of the function cannot be assigned. |
| 200 | Option not part of this config. | |
| 201 | End of memory. | Not enough space for program and data in the main storage. |
| 202 | Protection violation. | |
| 203 | Incompatible SAVE format. | The program is saved under an incompatible Monroe BASIC version. |
| 204 | Can't merge compiled programs. | |
| 205 | GRAPHICS is too late to COMMON. | COMMON statement error. |
| 206 | Please use the RUN command. | |
| 207 | Can't continue. | Applies to GOTO line number and CON. |

| Number | Message | Meaning |
|---|---|---|
| 208 | Illegal direct mode. | The instruction cannot be used as a command. |
| 209 | Bad command argument. | Wrong argument to the command e.g. LIST ##. |
| 210 | Illegal number. | The number contains other characters than digits. |
| 211 | Can't change precision. | Change of precision after assignment not allowed. |
| 212 | Compiler buffer overflow. | Line longer than 160 characters. |
| 220 | Undecodeable statement. | Formal Monroe BASIC error. |
| 221 | Text expected after end of line. | Formal Monroe BASIC error. Statement syntax not complete. |
| 222 | Must be first on a line. | |
| 223 | Illegal index or arguments. | |
| 224 | Illegal mode mixing in expression. | |
| 225 | Not a simple variable. | Indexed variable not allowed e.g. in a FOR loop. |
| 226 | Illegal statement after ON. | Formal Monroe BASIC error. |
| 227 | Can't find comma. | |
| 228 | Can't find '=' | |
| 229 | Can't find ')' | Formal Monroe BASIC error. |
| 230 | Can't find AS FILE | Missing in OPEN and PREPARE instructions. |
| 231 | Can't find AS. | Error in NAME ...A$ ... . |
| 232 | Can't find TO. | In FOR loops. |
| 233 | Line number expected. | Illegal variable name. |
| 234 | Illegal variable name. | |

# APPENDIX C
# SAMPLE PROGRAMS

# APPENDIX C
# SAMPLE PROGRAMS

This Appendix contains the following program examples:

1. Create File Containing Fixed Length Records.
2. Run Utility Program from Monroe BASIC Program.
3. Multi-tasking.

## C.1 CREATE FILE CONTAINING FIXED LENGTH RECORDS

```
5   ! FOR REFERENCES PURPOSES THIS PROGRAM IS NAMED FIXLEN
10  ! HOW TO ALLOCATE A FILE WITH FIX RECORD LENGTH
20  !
30  DIM A%(8%), B%(6%)
40  DIM A$=29%
50  ;
60  ; "** THIS PROGRAM CREATES A FILE WITH FIX RECORD LENGTH **"
70  ;
80  INPUT "NAME OF THE FILE TO BE CREATED? " B$
90  INPUT "RECORD LENGTH ?  "R%
100 !
110 ! SUPERVISOR CALL SVC2.3 IS USED TO SET THE CORRECT (UNPACK)
120 ! FORMAT OF THE FILE NAME.
130 !
140 B%(0%)=1%
150 B%(1%)=3%
160 B%(2%)=VARPTR(B$)
170 B%(3%)=VARPTR(A$)+1%
180 SVC 2%,B%
190 !
200 ! SUPERVISOR CALL SVC 7 IS USED TO ALLOCATE THE FILE WITH
210 ! DESIRABLE RECORD LENGTH (R%)
220 !
230 A%(0%)=1%
240 A%(1%)=256%*16%+1%
250 A%(2%)=VARPTR(A$)+1%
260 A%(4%)=R%
270 SVC 7%,A%
280 IF ERRCODE=0 THEN 300
290 STOP
300 ; : ; "THE FILE ***  "; : ; B$; : ; " *** HAS BEEN CREATED,"
310 ; "WITH RECORD LENGTH OF *** "; : ; R%; : ; " ***."
320 END
```

## C.2 RUN UTILITY PROGRAM FROM MONROE BASIC PROGRAM

```
10  !
20  ! SIMPLE EXAMPLE SHOWING HOW TO START A UTILITY PROGRAM
30  ! FROM A BASIC PROGRAM USING SVC6.
40  !
50  DIM A%(7%) ! DIMENSION SVC 6 BLOCK.
60  !
70  ! START COPYLIB PROGRAM.  NAME SHOULD HAVE UNPACKED
75  ! FORMAT AS DESCRIBED UNDER SVC 2.3
80  !
84  ! A$ IS CONSTRUCTED AS FOLLOWS:
85  ! A$=|VOL(4 CHRS.)|FILENAME(12 CHRS.)|DIRECTORYNAME(12CHRS.)|
86  ! =28 CHRS.
90  A$="    COPYLIB      "+SPACE$(12%)
95  INPUT "COPY FROM VOLUME? "G$
97  INPUT "COPY TO VOLUME? ",Q$
98  INPUT "FILE TO BE COPIED (ADD COMMA AFTER NAME)? "F$
100 B$=G$+F$+Q$+CHR$(0,0) ! PARAMETER STRING
102 B$=CVT%$(LEN(B$) AND 254)+B$ ! LENGTH OF STRING MUST BE FIRST IN PAR.
103 ! STRING
104 C$="COPY"! THIS WILL BE THE TASK-ID.
105 !
106 ! START CREATE SVC BLOCK.
107 !
110 A%(0%)=1%+2% ! FUNCTION CODE LOAD AND START TASK
120 A%(2%)=VARPTR(C$) ! TASK-ID NAME.
130 A%(3%)=VARPTR(B$) ! POINTER TO PARAMETER STRING.
135 A%(5%)=VARPTR(A$) ! POINTER TO TASK TO BE STARTED.
140 A%(6%)=10000% ! ADD EXTRA SIZE FOR FASTER COPYING.
150 SVC 6%,A%,0%,64% ! USE SVC6 TO START TASK WITH OPTION SWITCH
155 ! 'G' SET.
160 !
170 ! WAIT FOR TASK TO BE FINISHED.
180 !
190 A%(0%)=32%+8%
200 SVC 6%,A%
210 !
215 ;
220 ; "ALL DONE !"
230 END
```

## C.3 MULTI-TASKING

The user can directly run more than one Monroe BASIC program simultaneously through the use of the LOAD and RUN statements. The following commands load and start a low priority BASIC spooler program (SPOOLER) as a second task, and a BASIC report generator program (REPORTGEN) as the primary task:

```
-LOAD BASIC,SPOL,8000¶          (task name, xmemory)
-PRIORITY SPOL,150¶             (task name, new priority)
-START SPOL,C SPOOLER¶          (task name, option program name)

-BASIC REPORTGEN¶
.
.
.
```

Note that the above commands could be contained in a command file invoked by the SETAUTO Utility. Refer to the 8800 Series Utility Programs Programmers Reference Manual for details on LOAD, RUN, etc.

This next example shows the sample program required to run two Monroe BASIC programs at the same time while in BASIC. These programs should not request input or direct output to the same device, console or printer; output can be sent to the same disk device, but different filenames must be used. The Monroe BASIC program (see PRINT, below) which is referenced in the calling program (see MULTITASKPRG, below) should contain a BYE statement as the last program line.

Before executing this program (see MULTITASKPRG, below) the user can run the Task SLICE to specify the amount of processor time to be given to each program. (Refer to the 8800 Series Utility Programs Programmer's Reference Manual for details.)

```
-SLICE 100¶ (Sets current slice to 100 milliseconds)
-BASIC¶
BASIC8 R1-nn yyyy-mm-dd
BASIC
```

```
LIST MULTITASKPRG¶
10  DIM Svcblk%(6%)
20  Load%=1%
30  Start%=2%
40  !
45  ! ********************************************************************
50  ! *
60  ! *          LOAD AND START THE OTHER PROGRAM
70  ! *
76  !
80  Progname$='     BASIC'+SPACE$(19%)
90  Tskname$='TEST' ! ASSIGNS 4-LETTER TASK NAME TO PRINT PROGRAM TASK
95  ! SYSTEM AUTOMATICALLY ASSIGNS 4-LETTER NAME OF USPO TO THIS
96  ! PROGRAM TASK
100  Par$='PRINT'+CHR$(0%,0%)
105  ! make sure the length is an even number
110  Par$=CVT%$(LEN(Par$) AND 254%)+Par$
120  Svcblk%(0%)=Load%+Start%
130  Svcblk%(1%)=0%
140  Svcblk%(2%)=VARPTR(Tskname$)
150  Svcblk%(3%)=VARPTR(Par$)
160  Svcblk%(4%)=0%
170  Svcblk%(5%)=VARPTR(Progname$)
180  Svcblk%(6%)=0%
190  ON ERROR GOTO 210
200  SVC 6%,Svcblk%
201  GOTO 250
210  ; 'ERROR ';INT(Svcblk%(0%)/256%)
220  STOP
230  !
235  ! ********************************************************************
240  ! *
241  ! *          CONTINUE YOUR WORK IN THIS PROGRAM
242  !
243  !
250  FOR I%=1% TO 100%
260    ; I%
270  NEXT I%
280  END
BASIC
```

```
LOAD PRINT¶
BASIC¶
LIST¶
10  OPEN 'PR:' AS FILE 1%
20  FOR I%=1% TO 40%
30    ; #1%,STRING$(40%,64%+I%)
40  NEXT I%
50  BYE
BASIC
```

```
RUN MULTITASKPRG¶
1  (Task USPO starts listing on console)
2  (Task Test starts listing on
3   printer)
4
5
6
hh.mm.ss  END OF TASK xx   (Task TEST ends)
7
8
.
.
.
100
(Task USPO ends)
BASIC
```

# APPENDIX D
# PORT NUMBER ASSIGNMENTS

# APPENDIX D
# PORT NUMBER ASSIGNMENTS

| Hardware | Port Number |
|---|---|
| COUNTER TIME CIRCUIT (CTC) | |
| Baud Rate Generator for Communication Port | 168 |
| Baud Rate Generator for RS232 Port | 169 |
| Baud Rate Generator for Printer Port | 170 |
| Real Time CLock | 171 |
| SERIAL I/O (SIO) | |
| Communication Port Data | 164 |
| Communication Port Commands | 165 |
| Auxiliary Port Data | 166 |
| Auxiliary Port commands | 167 |
| MAP | |
| Program Map for Segment A | 196 |
| Program Map for Segment B | 197 |
| DMA Map for Segment A | 198 |
| DMA Map for Segment B | 199 |
| SWITCH SET | 255 |
| SYS | 200 |
| COLOR PROM (HC) | 212 |
| VIDEO OUTPUT ADD (HS) | 216 |
| FLOPPY CONTROLLER (1793) | |
| Floppy Controller Status/Command | 176 |
| Floppy Controller Track | 177 |
| Floppy Controller Sector | 178 |
| Floppy Controller Data | 179 |
| PARALLEL I/O (PIO) | |
| General Status Bits Port Data | 180 |
| General Status Bits Port Command | 181 |
| Uncommitted Parallel Port Data | 182 |
| Uncommitted Parallel Port Command | 183 |
| FLOP | 192 |
| DART | |
| Printer Port Data | 160 |
| Printer Port Command | 161 |
| Keyboard Port Data | 162 |
| Keyboard Port Command | 163 |

| Hardware | Port Number |
|---|---|
| DMA | 172 |
| SOUND | 204 |
| NONMASKABLE INTERRUPT (NMIOFF) | 220 |
| BUSINESS VIDEO | |
| Write Register Select | 184 |
| Write Data | 185 |
| Not Used | 186 |
| Read Data | 187 |
| READ STROBE NE558 (EDUCATION) | 184,185 |
| READ JOYSTICK PORT (EDUCATION) | 186,187 |

# APPENDIX E
# LOW RESOLUTION COLOR GRAPHICS CHARACTER SET

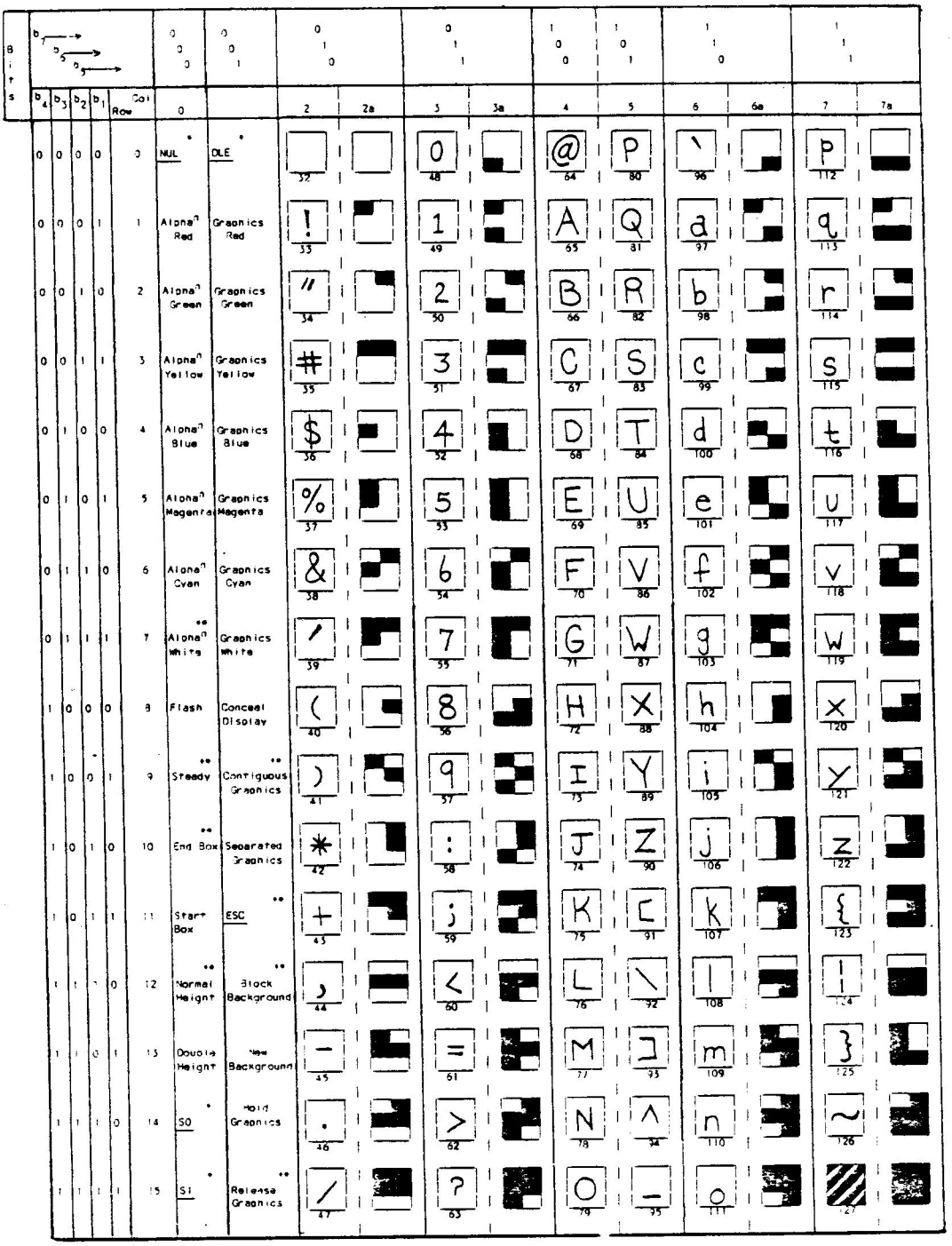
# APPENDIX E
# LOW RESOLUTION COLOR
# GRAPHICS CHARACTER SET

Table E-1 shows the low resolution color graphics character set. Each of the 64 graphics characters in Table B-1 (columns 2a, 3a, 6a and 7a) can be in any of the seven standard colors above. Graphics characters are displayed on a 2-by-3 matrix. Six bits determine which cells on the matrix are illuminated, while the seventh bit ($b_6$) distinguishes between alphanumeric and graphics characters.

If $b_6$ is a '0', then the code is always for an alphanumeric character; if it is a '1", then the code is either for an alphanumeric (columns 2,3,6 and 7) or a graphic character (columns 2a, 3a, 6a and 7a); the control characters determine which it is.

Control characters shown in columns 0 and 1 are normally displayed as spaces. Codes may be referred to by their column and row e.g. 2/5 refers to 26. Black represents display color. White represents background color.

Table E-1. Low Resolution Color Graphics Character Set




* These control characters are reserved for compatibility with other data codes.

** These control characters are presumed before each row begins.

# APPENDIX F
# HIGH RESOLUTION
# COLOR SELECTION CHART

# APPENDIX F
# HIGH RESOLUTION
# COLOR SELECTION CHART

The color selection statement, FGCTL, selects the 2 or 4 color combination to be in effect. This is done by specifying a number with this statement between 0 and 255. Table F-1 lists these numbers and their associated colors. Numbers less than 128 indicate that the ordinary text and graphics are mixed with the high resolution graphics. From 128 to 255 only the high resolution graphics memory will be displayed. Note that after executing a program specifying FGCTL 128 to 255, a LIST command must be entered to get back to BASIC. Combinations 72 to 127 and 200 to 255 are used in the animation mode.

If a black and white monitor is used, color 3 will always be white and color 0 will always be black. To indicate white for color 1 and 2, the symbol † will be appended to either:

1. the specific color in the table (for color combinations 72 to 127).

   or

2. the top of the table when it refers to all items for that color number (for color combinations 1 to 71).

Table F-1. High Resolution Color Section Table

| Selection Number | | Colors* | | | |
|---|---|---|---|---|---|
| Graphics Text | High Resolution Only | 0 | 1† | 2† | 3† |
| 0 | 128 | BK | BK | BK | BK |
| 1 | 129 | BK | W | W | W |
| 2 | 130 | BK | R | GR | Y |
| 3 | 131 | BK | R | GR | B |
| 4 | 132 | BK | R | GR | M |
| 5 | 133 | BK | R | GR | C |
| 6 | 134 | BK | R | GR | W |
| 7 | 135 | BK | R | Y | B |
| 8 | 136 | BK | R | Y | M |
| 9 | 137 | BK | R | Y | C |
| 10 | 138 | BK | R | Y | W |
| 11 | 139 | BK | R | B | M |
| 12 | 140 | BK | R | B | C |
| 13 | 141 | BK | R | B | W |
| 14 | 142 | BK | R | M | C |
| 15 | 143 | BK | R | M | W |
| 16 | 144 | BK | R | C | W |
| 17 | 145 | BK | GR | Y | B |
| 18 | 146 | BK | GR | Y | M |
| 19 | 147 | BK | GR | Y | C |
| 20 | 148 | BK | GR | Y | W |
| 21 | 149 | BK | GR | B | M |
| 22 | 150 | BK | GR | B | C |
| 23 | 151 | BK | GR | B | W |
| 24 | 152 | BK | GR | M | C |
| 25 | 153 | BK | GR | M | W |
| 26 | 154 | BK | GR | BK | W |
| 27 | 155 | BK | Y | B | M |
| 28 | 156 | BK | Y | B | C |

* B=blue, C=cyan, Y=yellow, GR=green, M=magenta, R=red, BK=black, W=white

† white on black/white monitor

## APPENDIX F - HIGH RESOLUTION COLOR SELECTION CHART

| Selection Number | | Colors* | | | |
|---|---|---|---|---|---|
| Graphics Text | High Resolution Only | 0 | 1† | 2† | 3† |
| 29 | 157 | BK | Y | B | W |
| 30 | 158 | BK | Y | M | C |
| 31 | 159 | BK | Y | M | W |
| 32 | 160 | BK | Y | C | W |
| 33 | 161 | BK | B | M | C |
| 34 | 162 | BK | B | M | W |
| 35 | 163 | BK | B | C | W |
| 36 | 164 | BK | M | C | W |
| 37 | 165 | R | GR | Y | B |
| 38 | 166 | R | GR | Y | , |
| 39 | 167 | R | GR | Y | C |
| 40 | 168 | R | GR | Y | W |
| 41 | 169 | R | GR | B | M |
| 42 | 170 | R | GR | B | C |
| 43 | 171 | R | GR | B | W |
| 44 | 172 | R | GR | M | C |
| 45 | 173 | R | GR | M | W |
| 46 | 174 | R | GR | C | W |
| 47 | 175 | R | Y | B | M |
| 48 | 176 | R | Y | B | C |
| 49 | 177 | R | Y | B | W |
| 50 | 178 | R | Y | M | C |
| 51 | 179 | R | Y | M | W |
| 52 | 180 | R | Y | C | W |
| 53 | 181 | R | B | M | C |
| 54 | 182 | R | B | M | W |
| 55 | 183 | R | B | C | W |
| 56 | 184 | R | M | C | M |
| 57 | 185 | GR | Y | B | M |

* B=blue, C=cyan, Y=yellow, GR=green, M=magenta, R=red, BK=black, W=white

† white on black/white monitor

## APPENDIX F - HIGH RESOLUTION COLOR SELECTION CHART

| Selection Number | | Colors* | | | |
|---|---|---|---|---|---|
| Graphics Text | High Resolution Only | 0 | 1† | 2† | 3† |
| 58 | 186 | GR | Y | B | C |
| 59 | 187 | GR | Y | B | W |
| 60 | 188 | GR | Y | M | C |
| 61 | 189 | GR | Y | M | W |
| 62 | 190 | GR | Y | C | W |
| 63 | 191 | GR | B | M | C |
| 64 | 192 | GR | B | M | W |
| 65 | 193 | GR | B | C | W |
| 66 | 194 | GR | M | C | W |
| 67 | 195 | Y | B | M | C |
| 68 | 196 | Y | B | M | W |
| 69 | 197 | Y | B | C | W |
| 70 | 198 | Y | M | C | W |
| 71 | 199 | B | M | C | W |
| | | 0 | 1 | 2 | 3† |
| 72 | 200 | BK | R† | BK | R |
| 73 | 201 | BK | BK | R† | R |
| 74 | 202 | BK | GR† | BK | GR |
| 75 | 203 | BK | BK | GR† | GR |
| 76 | 204 | BK | Y† | BK | Y |
| 77 | 205 | BK | BK | Y† | Y |
| 78 | 206 | BK | B† | BK | B |
| 79 | 207 | BK | BK | B† | B |
| 80 | 208 | BK | M† | BK | M |
| 81 | 209 | BK | BK | M† | M |
| 82 | 210 | BK | C† | BK | C |
| 83 | 211 | BK | BK | C† | C |
| 84 | 212 | BK | W† | BK | W |
| 85 | 213 | BK | BK | W† | W |
| 86 | 214 | R | GR† | R | GR |

* B=blue, C=cyan, Y=yellow, GR=green, M=magenta, R=red, BK=black, W=white

† white on black/white monitor

## APPENDIX F - HIGH RESOLUTION COLOR SELECTION CHART

| Selection Number | | Colors* | | | |
|---|---|---|---|---|---|
| Graphics Text | High Resolution Only | 0 | 1 | 2 | 3† |
| 87 | 215 | R | R | GR† | GR |
| 88 | 216 | R | Y† | R | Y |
| 89 | 217 | R | R | Y† | Y |
| 90 | 218 | R | B† | R | B |
| 91 | 219 | R | R | B† | B |
| 92 | 220 | R | M† | R | M |
| 93 | 221 | R | R | M† | M |
| 94 | 222 | R | C† | R | C |
| 95 | 223 | R | R | C† | C |
| 96 | 224 | R | W† | R | W |
| 97 | 225 | R | R | W† | W |
| 98 | 226 | GR | Y† | GR | Y |
| 99 | 227 | GR | GR | Y† | Y |
| 100 | 228 | GR | B† | GR | B |
| 101 | 229 | GR | GR | B† | B |
| 102 | 230 | GR | M† | GR | M |
| 103 | 231 | GR | GR | M† | M |
| 104 | 232 | GR | C† | GR | C |
| 105 | 233 | GR | GR | C† | C |
| 106 | 234 | GR | W† | GR | W |
| 107 | 235 | GR | GR | W† | W |
| 108 | 236 | Y | B† | Y | B |
| 109 | 237 | Y | Y | B† | B |
| 110 | 238 | Y | M† | Y | M |
| 111 | 239 | Y | Y | M† | M |
| 112 | 240 | Y | C† | Y | C |
| 113 | 241 | Y | Y | C† | C |
| 114 | 242 | Y | W† | Y | W |
| 115 | 243 | Y | Y | W† | W |
| 116 | 244 | B | M† | B | M |
| 117 | 245 | B | B | M† | M |

* B=blue, C=cyan, Y=yellow, GR=green, M=magenta, R=red, BK=black, W=white

† white on black/white monitor

## APPENDIX F - HIGH RESOLUTION COLOR SELECTION CHART

| Selection Number | | Colors* | | | |
|---|---|---|---|---|---|
| Graphics Text | High Resolution Only | 0 | 1 | 2 | 3† |
| 118 | 246 | B | C† | B | C |
| 119 | 247 | B | B | C† | C |
| 120 | 248 | B | W† | B | W |
| 121 | 249 | B | B | W† | W |
| 122 | 250 | M | C† | M | C |
| 123 | 252 | M | M | C† | C |
| 124 | 253 | M | W† | M | W |
| 125 | 254 | M | M | W† | W |
| 126 | 255 | C | W† | C | W |
| 127 | 256 | C | C | W† | W |

* B=blue, C=cyan, Y=yellow, GR=green, M=magenta, R=red, BK=black, W=white

† white on black/white monitor

# APPENDIX G
# QUICK REFERENCE SUMMARY

# APPENDIX G
# QUICK REFERENCE SUMMARY

| Reference & Format | Use | Page |
|---|---|---|
| ABS(x) | Returns absolute value of x. | 10-4 |
| ADD$(A$,B$,p%) | Returns the addition of two strings. | 10-24 |
| ASCII(A$)<br>ASC(A$) | Returns the ASCII value of first character of A$. | 10-25 |
| ATN(x) | Returns the arctangent (in radians) of x. | 10-5 |
| AUTO [line no.] [,incr] | Automatic line numbering. | 6-4 |
| BYE | Transfers control to operating system. | 9-4 |
| CALL(A$[,D%]) | Calls an assembler program. CALL can destroy program execution if used erroneously. | 14-3 |
| CHAIN <string> | Loads and executes a program. | 9-5 |
| CHR$(m1[,m2,m3,...]) | Returns a character string corresponding to the ASCII values of the arguments. | 10-26 |
| CLEAR | Clears all variables and closes all open files. | 6-6 |
| CLOSE [channel no,...] | Closes the file(s). | 8-2 |
| COMMON <list> | Declares the variables, whose values are to be transferred to another program. | 9-7 |
| COMP%(A$,B$) | Returns a truth value based on a comparison two numeric strings. | 10-27 |
| CON (or CONT) | Continues program execution. | 6-7 |
| COS(x) | Returns the cosine of the x (x is in radians). | 10-6 |
| CUR(<y,x>) | Moves the cursor to line y%, position x%. | 10-41 |
| CURREAD <ypos,xpos> | Reads the current cursor position. | |
| CVTF$(n) | Returns a four- or eight-character string representation of a floating point number. | 14-9 |
| CVT$F(string) | Returns the floating point number representation of the first four or eight characters of a string. | 14-10 |
| CVT$%(string) | Returns the integer representation of the first two characters of a binary string. | 14-7 |

| Reference & Format | Use | Page |
|---|---|---|
| CVT%$(integer var.) | Returns a two-character string representation of an integer. | 14-5 |
| DATA <list> | Assigns values to variables (used with READ). | 7-3 |
| DEF FN<name> [type] [(argument)]=<expression> | Defines a single line function. | 9-8 |
| DEF FN<name> [type] [(arguments)] [(LOCAL variable,variable,...)] | Defines a multiple line function. | 9-8 |
| DIGITS <number> | Specifies the number of digits to be printed. | 8-3 |
| DIM <var list> or<br>DIM <string var=expr> | Allocates space for strings and vectors. | 7-5 |
| DIV$(A$,B$,p%) | Returns the quotient A$/B$ rounded off to (+) p% decimals or to (-) p places of precision. | 10-28 |
| DOUBLE | Sets floating point numbers to double precision (16 digits) mode. | 7-8 |
| ED [line no.] | Starts program editing. | 6-8 |
| END | Terminates the program. | 9-12 |
| ERASE <argument> | Erases one or more program lines. | 6-10 |
| ERRCODE | Returns the value of the latest generated error code. | 10-42 |
| EXP(x) | Returns the value $e^x$. | 10-7 |
| EXTEND | Allows extended variable names to be used. | 7-9 |
| FGCIRCLE | Draws a circle. | 13-4 |
| FGCTL <number> | Selects the color four-color combination in effect. | 13-10 |
| FGDRAW | Displays a specified shape. | 13-11 |
| FGERASE | Sets all elements of a shape to its background color. | 13-18 |
| FGFILL x,y[,color number] | Fills a rectangle from the previous position to the position indicated by the coordinates (x,y). | 13-19 |

| Reference & Format | Use | Page |
|---|---|---|
| FGGET | Copies a rectangle. | 13-6 |
| FGLINE x,y[,color number] | Draws a line from the previous position to the position indicated by the coordinates (x,y). | 13-21 |
| FGPAINT[x,y[,color number]] | Fills a closed area. | 13-24 |
| FGPOINT x,y [,color number] | Turns on a pixel on line x (0-239) in position y (0-230). | 13-27 |
| FGPOINT (x,y) | Returns color number of specified pixel. | 13-27 |
| FGPUT <string var> | Restores to high res memory the contents of the specified string variable. | 13-30 |
| FGROT <number> | Specifies the degree of rotation. | 13-32 |
| FGSCALE <x,y> | Scales the coordinates of the shape to be displayed. | 13-33 |
| FIX(x) | Returns the truncated value of x. | 10-8 |
| FLOAT | Specifies that all numbers will be interpreted as floating point. | 7-10 |
| FN<name> [type] [(parameter)] | Calls a user defined function. | 10-43 |
| FNEND | Terminates a multiple line function. | 9-14 |
| FOR <var>=<expr> to <expr> [STEP expr] | Starts a program loop. | 9-1 |
| GET <string variable> | Reads one or more characters from the keyboard. | 8-4 |
| GET #<channel no.>,<string var> [COUNT number] | Reads from a file. | 8-4 |
| GOSUB <line no.> | Unconditional jump to a subroutine. | 9-17 |
| GOTO <line no.> | Unconditional jump to the given line number. | 9-19 |
| HEX$(x) | Returns the hexadecimal string representation of a decimal number. | 10-9 |

| Reference & Format | Use | Page |
|---|---|---|
| IF <condition> [THEN or GOTO] <arg1> [ELSE arg2] | Conditional control of the order of execution of the program lines. | 9-20 |
| INP(i%) | Returns the data value from the in-port i%. | 14-11 |
| INPUT [#channel no.] <list> | Fetches data for the current program. | 8-6 |
| INPUT LINE [#channel no.,]<string variable> | Accepts a line of characters. | 8-9 |
| INSTR(n%,A$,B$) | Returns the position of string B$ in A$ starting search at position n%. | 10-29 |
| INT(x) | Returns the value of the greatest integer less than or equal to x. | 10-10 |
| INTEGER | Specifies that all variables are supposed to be integer variables, unless otherwise declared. | 7-12 |
| KILL <string> | Erases the file in question from external storage. | 8-11 |
| LEFT[$]A$,i%) | Returns the first i% characters of the string A$. | 10-30 |
| LEN(A$) | Returns the string length (including spaces) of A$. | 10-31 |
| [LET] <var> = <expr> | Assigns a value to a variable. | 7-4 |
| LIST [arguments] | Lists, saves or prints a program. | 6-11 |
| LOAD <fd> | Loads a program into working storage of the computer. | 6-13 |
| LOG(x) | Returns the natural logarithm of x. | 10-12 |
| LOG10(x) | Returns the common logarithm of x. | 10-13 |
| MERGE <fd> | Merges program files. | 6-15 |
| MID[$](A$,p%,k%)<br>[LET] MID[$](A$,p%,k%) = <expr> | Returns or replaces the substring of A4, which starts in position p% and has a length of k% characters. | 10-32 |
| MOD(<argument1>,<argument2>) | Returns the remainder of an integer division of the arguments. | 10-14 |

| Reference & Format | Use | Page |
|---|---|---|
| MUL$(A$,B$,p%) | Returns the product A$*B$ with p% (+) decimals or with p (-) places of precision. | 10-33 |
| NAME <string1> AS <string2> | Changes the name of a file. | 8-13 |
| NEW | Clears storage. | 6-17 |
| NEXT <variable> | NEXT terminates a program loop, which begins with a FOR statement. | 9-24 |
| NO EXTEND | Terminates work in EXTEND mode. | 7-15 |
| NO TRACE | Terminates the printout of line numbers, which was started by the instruction TRACE. | 9-25 |
| NUM$(argument) | Returns the numeric string corresponding to the argument. | 10-34 |
| OCT$(argument) | Returns an octal string representation of a decimal number. | 10-15 |
| ON ERROR GOTO [line number] | Branches to the indicated line number of an error. | 9-26 |
| ON <expression> GOSUB <line no.>[,line no.,...] | Conditional jump to one of several subroutines or to one of several entry points in a subroutine. | 9-28 |
| ON <expr> GOTO <line no.>[,line no.,...] | Jump to one of several line numbers, depending on the value of the expression. | 9-30 |
| ON <expr> RESTORE <line no.>[,line no.,...] | Sets the DATA pointer by the same selection routine as ON - GOTO. | 9-31 |
| ON <expr> RESUME <line no.>[,line no.,...] | Jump to one of several line numbers, depending on the value of the expression. The error handling is resumed. Used with ON ERROR GOTO. | 9-32 |
| OPEN <string> AS FILE <expr> [MODE expr] | Opens a file. | 8-15<br>14-24 |
| OPTION BASE <n> | Denotes the default minimum subscript value. | 7-16 |

| Reference & Format | Use | Page |
|---|---|---|
| OPTION EUROPE <n> | Specifies European or American PRINT USING "!" and "." field parameters. | 8-18 |
| OUT <port,data> [port,data,...] | Addresses the out ports at data output. | 14-27 |
| PAUSE | Pauses the current program task. | 9-33 |
| PDL(<argument>) | Returns joysticks x or y coordinate. | 30-45 |
| PEEK(<i%>) | Returns the contents of one byte at storage address i5. | 14-29 |
| PEEK2(<b%>) | Returns the contents of two bytes at storage address b%. This function is meant for advanced programming. | 14-30 |
| PI | Returns a constant value 3.14159 (single precision). | 10-16 |
| POKE <addr,data>[,data,...] | Loads a value into storage cell. | 14-31 |
| POSIT #<channel no.>[,number] | Positions the file pointer. | 8-19 |
| POSIT (<channel no.>) | Returns position of file pointer. | 14-32 |
| PREPARE <string> AS FILE <expr> [MODE expr] | Creates and opens a new file. | 8-21 |
| PRINT [#channel no.;]<argument>[,argument,...] | Prints data in ASCII format. | 8-23 |
| PRINT USING <format string><list,...] | Prints numbers and strings with the specified format. | 8-26 |
| PUT [#<channel no.>],<string expr> | Writes a string variable in binary format. | 8-27 |
| RANDOMIZE | Sets a random starting value for the RND function (the random number generator). | 7-17 |
| READ <variable>[,variable,...] | Used together with DATA statements as a way of assigning values to variables. | 7-19 |

| Reference & Format | Use | Page |
|---|---|---|
| REM text | Inserts comments in a program. | 10-47 |
| REN [<line no.1>[,incr[line no.2-line no.3]]] | Changes the line numbering of the current program. | 6-18 |
| RESTORE <line number> | Makes possible renewed use of the contents of DATA statements. | 7-21 |
| RESUME <line number> | Returns from error handler. | 9-34 |
| RETURN [variable] | Returns from subroutine or multiple line functon. | 9-35 |
| RIGHT[$](A$,n%) | Returns the last characters of A$ starting at position n%. | 10-35 |
| RND | Returns a random number between 0 and 0.999999. | 10-17 |
| RUN <fd> | Loads and executes a Monroe BASIC program or executes the current program. | 6-20 |
| SAVE <fd> | Creates a disk file and stores the current program into that file. | 6-22 |
| SCR | Clear storage. | 6-24 |
| SET TIME <string> | Sets the system's time and date. | 7-22 |
| SGN(x) | Returns the value +1 if x is positive, 0 if x and -1 if x is negative. | 10-18 |
| SIN(x) | Returns the Sine of x (x is in radians). | 10-19 |
| SINGLE | Changes all variables and expressions, which are floating point numbers, to single precision (6 digits). | 7-23 |
| SLEEP <expr> | Suspends currently running program for a specified number of seconds. | 10-45 |
| SOUND <channel%><pitch%>,<atten%> | Returns sounds on system speakers with specified qualities. | 10-49 |
| SPACE$(n%) | Returns a string consisting of n% spaces. | 10-36 |

| Reference & Format | Use | Page |
|---|---|---|
| SQR(x) | Returns the square root of x. | 10-20 |
| STOP | Stops the program execution. | 9-37 |
| STRING$(I%,K%) | Returns a stirng of ASCII characters. | 10-37 |
| SUB$(A$,B$,p%) | Returns the arithmetic difference A$-B$ of the numeric strings A$ and B$ with (+) p% decimals or with (-) p places of precision. | 10-38 |
| SVC <x%>,<A%>[,b%][,d%] | Communicates with the operating system to perform special functions. | 14-37 |
| SWAP%(n%) | Returns integer with first and the second bytes of n% transposed. | 14-41 |
| SYS(i%) | Returns system status as follows:<br>SYS(2) Returns total space available for program.<br>SYS(3) Program size.<br>SYS(4) Remaining storage space.<br>SYS(5) Keyboard input flag.<br>SYS(6) Puts back the last input character into the keyboard buffer.<br>SYS(7) ASCII key value of the terminating key used after the last INPUT or INPUT LINE statement.<br>SYS(10) Points to information block about the program.<br>SYS(11) Starting address of the program.<br>SYS(12) Gives a pointer to the variable root for all variables in Monroe BASIC. | 14-42 |
| TAB(i%) | Tabulates to the i%-th position on the line. | 10-51 |
| TAN(x) | Returns the tangent of x (x in radians). | 10-21 |
| TIME$ | Returns year-month-day hours.min.sec initially set by SET TIME. | 10-52 |
| TRACE [#channel no.] | Prints the line number of the executed program lines. | 9-38 |

| Reference & Format | Use | Page |
|---|---|---|
| TXPOINT (x,y)<br>TXPOINT x,y[,0] | Returns the value of or turns a graphical point on line y in position x on or off (with 0 included). | 12-20<br>12-21 |
| UNSAVE <fd> | Erases a file from a disk. | 6-25 |
| VAL(A$) | Returns the numeric value of the numeric string A$. | 10-39 |
| VAROOT(variable) | Returns the address of a table, which contains information about a variable. | 14-44 |
| VARPTR(Variable) | Returns the address of the value of a variable. | 14-44 |
| WEND | WEND terminates a loop that beings with WHILE. | 9-39 |
| WHILE <expression> | Specifies the conditions for branching out of a program loop. | 9-40 |

# INDEX

A



B



C