SOFTECH MICROSYSTEMS

A SUBSIDIARY OF SOFTECH

UCSD p-SYSTEM

A PRODUCT FOR MINI- AND MICRO-COMPUTERS

Versions II.0 and IV.0

BASIC REFERENCE MANUAL

First edition: February 1981

9494 Black Mountain Rd., San Diego, CA 92126

TWX: 910-335-1594
(714) 578-6105

DISCLAIMER:

This document and the software it describes are subject to change without notice. No warranty expressed or implied covers their use. Neither the manufacturer nor the seller is responsible or liable for any consequences of their use.

ACKNOWLEDGEMENTS:

This document was written by Stan Stringfellow at SofTech Microsystems. Special thanks are due to Dan LaDage of LaDage Computer Systems, as well as Gail Anderson, Blake Berry, and Randy Clark of SofTech Microsystems, for providing information and assistance. Also, thanks are due to Texas Instruments Incorporated, for providing useful information.

TABLE OF CONTENTS

7 FILE I/O AND VIRTUAL ARRAYS

CHAPTER 1

THE UCSD p-SYSTEM AND SOFTECH MICROSYSTEM BASIC

1.1 Introduction

SofTech Microsystems' BASIC is a compiled BASIC that runs under the UCSD p-System. Since it was designed to be used with the Screen Oriented Editor, it has an expanded syntax that allows indented and unnumbered statements as well as statements which are not in numeric order by line number. Because it is intended to be one language in a multi-language software development environment, BASIC subroutines can be separately compiled and linked into Pascal, FORTRAN, and BASIC host programs without recompilation. Also, BASIC programs may host separately compiled Pascal procedures and FORTRAN subroutines.

SofTech Microsystem's BASIC allows arrays with unlimited dimensions and subroutines with unlimited numbers of parameters. Virtual arrays which reside on disk, and may be very large, are permitted. Large programs may be split into many disk files, which can include each other using the $1 compile option, and be compiled into a single codefile.

The following sections describe how to use the UCSD p-System to create and compile BASIC programs. The compiler options are described, and the simple BASIC constructs relating to the text of the program itself are explained.

1.2 Editing BASIC Programs

The UCSD p-System Screen Oriented Editor is used to create and modify BASIC programs. This section will give a cursory explanation of how to use this Editor. For a more complete description of the Screen Oriented Editor, see the UCSD p-System Users' Manual.

To enter the Editor from the main system promptline, type "E". The system will respond by asking what file is to be edited. An existing text file may be specified (e.g., #5:PROG<return> will indicate the file PROG.TEXT on the disk in drive #5:) or a new file may be created by simply typing <return>.

Once the Editor has been entered, text may be inserted after typing an "I". A new line of text will automatically be indented to correspond to the line above it. This makes it easy to use the indentation feature of SofTech Microsystem's BASIC to increase readability:

```
        FOR I=1 TO 100
            FOR J=1 TO 100
                A(I,J)=0
                B(I,J)=J
            NEXT J
            IF I >= 50 THEN GOTO 10
            DISPLAY "I < 50"
10      NEXT I
        END
```

Note that line numbers are optional, and are really only necessary if a statement is to be the target of a GOTO or GOSUB statement (described in Chapter 4). When line numbers are used, they do NOT need to appear in increasing order.

Once the program has been typed in, the ETX key is typed to accept the text, and the Editor is exited by typing "Q" for quit. Then the user may U(pdate or W(rite the file. In the first case, the file will be saved as SYSTEM.WRK.TEXT and in the second case a name may be specified (e.g., #5:PROG<return> will save the file as PROG.TEXT on the disk in drive #5:).

1.3 Compiling BASIC Programs

Before the BASIC compiler can be invoked, the following steps must be followed:

1. Enter the Filer (by typing "F") and use the C(hange command to change the names of SYSTEM.COMPILER and SYSTEM.LIBRARY to some other names (such as PASCAL.COMPILER and SAVE.LIBRARY).

2. Change the names of BASIC.CODE and BASIC.LIBRARY to SYSTEM.COMPILER and SYSTEM.LIBRARY. If the new SYSTEM.LIBRARY is not on the * system disk, it must be T(ransfered there.

It may be desirable to create an entirely separate system disk to be used only with the BASIC language.

When the BASIC compiler is properly set up, it may be invoked from the main system promptline by typing "C". If the program text was U(pdated from the Editor, it will be automatically compiled.

If the program was W(ritten to a file (and no SYSTEM.WRK.TEXT already existed), the system will prompt for the name of the file to be compiled. The user should respond, for example #5:PROG<return> if the file is PROG.TEXT on #5. Then the system will prompt for the name of the codefile to be produced. The response

should be #5:PROG<return> if PROG.CODE on #5 is to be created.

At this point, the two pass compiler will execute, printing a dot for each line in the text. After compilation, "R" (for run) should be typed, and the program will execute (usually after linking the object with code from the Library).

If a syntax error is encountered by the compiler, the bell will sound, compilation will temporarily halt, and an error message will be displayed. The editor may be re-entered at this point by typing "E", and the error may be corrected immediately. Compilation may also be either continued or aborted.

Other errors, such as runtime arithmetic overflows, will be caught during program execution.

1.4 Compiler Options

The Compiler Options are used to control various aspects of the BASIC compiler's output. These options are specified in the following manner:

```
(*$<option> <parameters> *)
```

The INCLUDE option indicates to the Compiler that the specified file is to be compiled as though it were placed directly in line within the current file. The following is an example of this directive:

```
(*$I #5:PROG2.TEXT *)
```

The LIST option causes the compiler to emit a compiled listing to the CONSOLE, PRINTER or specified disk file:

```
(*$L CONSOLE: *)
(*$L PRINTER: *)
(*$L #4:LIST.TEXT *)
```

The listing can be optionally turned on or off at any point in the source text (after it has been started using one of the above forms of this directive) by following the directive with a "+" for on, or a "-" for off:

```
(*$L+*)
(*$L-*)
```

The PAGE option causes a form feed in a compiled listing at the point where it occurs in the source text:

```
(*$P*)
```

The FLIP option is used only in the II.0 version of the compiler. This causes the byte sex of the object to be of opposite sex from the host machine:

```
(*$F*)
```

The QUIET option controls the "quiet/noisy" mode of the compiler. In the noisy mode, $Q-, the compiler displays a dot (period) on the system console for each statement compiled. In quiet mode, $Q-, the dots are not displayed. The default is $Q- unless the machine has a "slow terminal" (designated by a data item in SYSTEM.MISCINFO), in which case the default is $Q+.

```
(*$Q+*)
(*$Q-*)
```

The RANGE CHECKING option controls runtime range checking on references to array variables and string variables. When $R+ is in effect, runtime range errors cause the program to abort with an execution error. If $R- is in effect, the compiler does not emit code to do range checking during execution. The default is $R+.

```
(*$R+*)
(*$R-*)
```

The I/O CHECK option directs the compiler to emit code which will cause I/O errors to be handled by the system if the $I+ option is on. The $I- option causes the I/O status to be recorded and made available, through the built in function IORESULT, but no execution error results from I/O errors. The default is $I+.

```
(*$I+*)
(*$I-*)
```

The T option, when $T+ is used, causes code to be emitted which handles the transcendentals in the library in a manner consistent with the TI machines. The default is T+.

```
(*$T+*)
(*$T-*)
```

The Copyright option will place a copyright notice within the codefile:

```
(*$C Copyright (c) 1981, SofTech Microsystems *)
```

1.5 Comment Delimiters

The REM statement and the exclamation point (!) are treated identically by the BASIC compiler. They represent the start of a comment which is terminated by the end of the line:

```
REM   This is a comment
A=1   REM This is a comment
! This is a comment
A=2   ! This is a comment
```

The two delimiters (* and *) can also be used to enclose comments. The comments between these two delimiters may cross line boundaries:

```
A=1   (* This is a comment *)
B=2   (* This is a comment: At this point we have
         decided to set B equal to 2 *)
```

1.6 Multiple Line Statements

Since placing an end of line between the delimiters (* and *) essentially causes that end of line to be invisible to the compiler, statements which are allowed to fill only one line may be expanded to several lines by commenting out the EOL character. For example, the following DIM statement (see Chapter 2) is used to declare two lines of arrays:

```
DIM   A(4), B(5,6,7), C(10,10), (*
      *) D(8,9), E(20)
```

And the following FUNCTION (see Chapter 6) is defined with more parameters than might fit on one line:

```
DEF A_FUNC(A,B,C,D,E,F,G,H,I,J,K, (*
           *) L,M,N,O,P,Q,R,S)
```

1.7 Multiple Statement Lines

Normally, only one statement is allowed on a line. The double colon (::) is used to separate statements so that two or more may appear on a single line as in the following example:

```
A=1 :: B=2 :: C=3
```

CHAPTER 2

DATA TYPES AND EXPRESSIONS

2.1 Data Types

SofTech Microsystems' BASIC handles both numeric and character string data types. Numeric data is expressed as INTEGER, REAL, or DECIMAL numbers. In the current version, the DECIMAL data type is identical to the REAL type. Arithmetic operations can be performed on this type of data. Character string data consists of sequences of printable ASCII characters. String operations may be performed on data of this type.

2.1.1 Numeric Data

Integers have no decimal point and are allowed to have values between 32767 and -32767.

Real or floating point numbers may have a decimal point and/or an 'E' followed by an exponent. If an exponent is specified, the floating point number will be raised to that power of ten. For example 2.0E7 is equivalent to 2.0 times ten to the seventh power. The minimum and maximum values for real numbers are machine dependent.

Decimal numbers are identical to real numbers. There are some syntax differences, but there is no difference at all in the way they are treated in the current version.

2.1.2 String Data

String data is nonnumeric information expressed as words or other character sequences. A string may contain numeric symbols, but arithmetic may not be performed on it. The string operations are described in Sections 2.7, 2.8, and Chapter 5.

2.2 Constants

Data may be in the form of constants. The value of a constant is specified at the time the program is written and does not change during program execution.

2.2.1 Numeric Constants

A numeric constant may be an integer or floating point number, and may be either positive or negative. The following are examples of numeric constants:

```
27
1981
123.4567
0.333
.333
333.0
10.0E3
10E3
-1
-765.4321
-0.001
-001.0
-12.234E5
-1E-15
```

The following are incorrect numeric constants:

25x2	The 'x' is not allowed
7,999.99	Commas are not allowed
10.0E2E3	Only one 'E' is allowed

2.2.2 String Constants

A string constant is a sequence of printable ASCII characters enclosed within double quotes. A quote may be inserted into a string by entering two consecutive quotes (""). The following are examples of string constants:

```
"Now is the time for all good men..."
"765.4321"
"<>,. ?/;: !@#$ %%&* ()'+ =_"
"Quoth the Raven, ""NEVER MORE!"""
```

The following are incorrect string constants:

'Incorrect'	Single quotes are not string delimiters
"WOW	Second quote missing
"She said, "Hi!""	Second quote is taken as end of string

2.3 Variables

Variables are data items which may have their values changed during the execution of a program. Like constants, variables may be numeric data or character strings. Variables may also be grouped into arrays. Within an array they may be accessed individually by specifying the array name and a subscript. For more information about arrays, see Section 2.4.

2.3.1 Numeric Variables

Numeric variables may be INTEGER, REAL, or DECIMAL format. The range of values which are valid for numeric variables is the same as for numeric constants. If, during program execution, an attempt is made to assign a variable to a value outside that range, a runtime error will result. (The exception to this is when a computed integer value overflows. An error will be produced only if the sign bit is changed due to an overflow. Other integer overflow errors are not detected.)

2.3.1.1 Numeric Variable Naming Conventions

Numeric variable names must begin with a letter of the alphabet. This letter may be followed by as many as 254 alpha-numeric characters or any of the special characters: @, [,], \ or _. All the characters in a variable name (up to 255) are used to distinguish it from other variables. The following are valid numeric variable names:

```
ONE
F[2]NUM
P_123
VERY_LONG_IDENTIFIERS_ARE_OK
L@PTR
```

Variable names may not be the same as reserved words used in SofTech Microsystems Basic. For example, a variable can not be named GOTO. A compiler error will result if this is attempted. A list of these reserved words appears in Appendix A.

2.3.1.2 Numeric Variable Declarations

All numeric variables are assumed to be of type REAL unless otherwise specified. The default type can be changed by using the ALL clause as shown below:

```
INTEGER ALL     Changes default type to INTEGER
REAL ALL        Changes default type REAL
DECIMAL ALL     Changes default type to DECIMAL
```

The ALL statement, if used, must precede the first occurrence of any of the following statements:

```
INTEGER
REAL
DECIMAL
DIM
DEF
SUB
```

Numeric variables can be individually declared to be of a particular type, regardless of what the default type is by using the INTEGER, REAL or DECIMAL statements. (Numeric variables can also be declared within the DIM statement, see section 2.4.2.) The type name is followed by a list of variables separated by commas as follows:

```
INTEGER I,J,K
REAL R
DECIMAL A1,A2,A3,A4,A5
DECIMAL (2) B1,B2,B3,B4,B5,B6
DECIMAL (-4) C1,C2,C3,C4,C5,C6,C7
```

The INTEGER declaration above specifies I, J, and K as integers. The REAL statement declares R as a REAL. The rest of the variables are declared to be DECIMAL numbers. The second and third DECIMAL statements contain an optional number in parentheses. In the current version, this number has no meaning, and DECIMAL statements are equivalent to REAL statements.

2.3.2 String Variables

String variables, like numeric variables, may have their values altered during program execution. A string may contain up to 255 printable ASCII characters. Strings are used to hold data for input and output and to express nonnumeric data such as names, descriptions, etc.

2.3.2.1 String Variable Naming Conventions

String variables are named according to the same conventions as numeric variables. The only difference is that string variable names must end with a dollar sign. The following are correct examples of string variables:

```
ASTRING$
ANOTHER_STRING$
S[22]$
A\B\C\$
```

2.3.2.2 String Declarations

Strings do not have to be declared, but declaring them may save memory space. If they are not declared, or if the declaration does not specify a size, they are allocated a default length of 255 bytes at compile time. This allocated space does not change dynamically during program execution. But a maximum size can be specified using the DIM statement:

```
DIM ASTRING$*20
```

This limits ASTRING$ to a maximum length of 20 bytes. For further information concerning the DIM statement in this context, see section 2.4.3.

2.4 Arrays

Variables may be grouped together into an ARRAY. The array is given a name, a number of dimensions, and a size for each dimension. By specifying the array name followed by the one or more index values, a specific variable within the array may be accessed. For example, a two dimensional array, AR1, would be refered to as AR1(4,3) in order to obtain the indicated element. An array may have any number of dimensions, but the total number of elements may not exceed 32767.

Virtual Arrays are arrays which reside on disk. This allows programs to use large arrays which will not fit into memory. This type of array is discussed in Chapter 7.

2.4.1 The DIM Statement

In order to declare an array, the DIM statement is used. This statement defines the number of dimensions and the number of entries within each dimension of the array.

In the declaration, DIM is followed by the array name. Then, in parentheses, one or more integers are specified, separated by commas. The array name should follow the conventions for numeric variable names if it is an array of numeric variables. Likewise, it should follow the conventions for string variable names if it is an array of strings.

The integers in parentheses are zero-based. (This may be changed, however, by the OPTION BASE statement. See Section 2.4.4.) This means that the array can be indexed from zero up to the number specified, and that the number of entries is one greater than that number.

More than one name may be defined in a DIM statement, if each name is separated by a comma.

The following are correct examples of array declarations:

```
DIM AR10 (9)
DIM ONE_ELEMENT_ARRAY(0)
DIM S\ARRAY$ (20,20)
DIM MULTI@ (3,7,15,31)
DIM A7(100,10,10),A8(100,10,10)
DIM LARGE(32000), W$(9,9,9), LIST_NUMS(0,17)
```

It is not always necessary to declare arrays. If the maximum dimension is less than or equal to 10 (e.g. DIM A(10,10,5)) then the array may be implicitly declared when it is first referenced. The default lower boundary for each dimension is 0 and the default upper boundary is 10. However, it may save considerable space to declare small multi-dimensional arrays anyway. For example, an integer array of dimension (2,2,2,2) would only take up 3 to the fourth (81) words of memory. If undeclared, however, this same array would default to a (10,10,10,10) dimensional array and take up 14641 words.

2.4.2 Type Specification of Numeric Arrays

All of the arrays declared in Section 2.4.1 above, except for S\ARRAY$ and W$, are numeric arrays. The variable type of the entries within those arrays will be the numeric variable default type (REAL unless otherwise specified by using the ALL statement). Numeric arrays may be declared to be of a particular type using either a DIM statement or an INTEGER, REAL, or DECIMAL statement as in the following examples:

```
DIM A(7,2), INTEGER B(2,3), C(12,1,0)
DIM REAL D(5,7), DECIMAL E(10,10,10), INTEGER F(0)
INTEGER I,J,K(7,7,7)
REAL    R(12),S
```

In the first line above, B is an array of type integer. A and C are arrays of the default type. In the second line, D is a real array, E is a decimal array, and F is an integer array. In the third line, I and J are integer variables and K is an integer array. The fourth line declares R to be a real array and S to be a real variable.

A DIM statement may also include single variables. These variables may optionally be preceded by a type declaration:

```
DIM A(99,9),B,C(49)
DIM E(4,5), INTEGER F1,F2, G(6,7)
DIM REAL H
```

The first line above declares arrays A and C and variable B. Both arrays and the

variable are of the default type. The second line declares arrays E and G to be of the default type. Variable F1 is declared to be an integer and variable F2 is declared as the default type. In the third line, real variable H is declared.

2.4.3 Size Specification of String Arrays

An array is declared as a string array by naming it according to string variable naming conventions:

```
DIM S_ARRAY$(2,3,4)
```

However each string in S_ARRAY$ above will consume 256 bytes of memory. A maximum size for each string in an array can be indicated by following the declaration with an asterisk (*) and a number between 1 and 255 inclusive. Also, string variables can be declared within DIM statements in the same way. The following are examples:

```
DIM S_ARRAY$(2,3,4)*20
DIM S1$(2)*10, S2$(20)*11
DIM A$(10,10)*1, B$*25, C$(5,6,7), D$*1
```

The first line above declares three-dimensional S_ARRAY$ to consist of strings with a maximum length of 20 bytes. The second line declares S1$ and S2$ as one-dimensional arrays containing strings with a maximum length of 10 and 11 bytes. In the third line, A$ is a 10 by 10 array of one character strings, B$ is a string variable with a maximum length of 25, C$ is a three dimensional string array of 256 byte strings, and D$ is a string variable containing at most one character.

This string length specification may occur anywhere that a string declaration is legal.

2.4.4 The OPTION BASE Statement

Array indices, whether declared in a DIM statement or not, are zero-based by default. This means that the statement, DIM A(10), declares A to be indexed from 0 to 10. By using the OPTION BASE statement, array indices can be based at 1 or 0:

```
OPTION BASE 0
OPTION BASE 1
```

The first statement leaves the indexing base at the default value of 0 and the second makes the base 1. The OPTION BASE statement may be used, at most, once in a program. If it is used, it must come before any statement which declares or references array elements. If 1 is declared to be the base, no statement may declare or reference an array with an index of zero.

2.5 The LET Statement

The LET statement is used to make numeric or string assignments. The word LET, which is optional, is followed by the variable to which a value is to be assigned. This is followed by an equal sign and an expression, the value of which will be assigned to the variable.

LET statement syntax:

```
LET numeric_variable = numeric_expression
LET string_variable  string_expression
numeric_variable = numeric_expression
string_varible = string_expression
```

LET statement examples:

```
LET A=1
LET R1=2.0 * (R1+1)
LET S1$="STRING"
LET S2$="ANOTHER "&S1$
A=2
R1=R2+1.0
S2$=S1$
```

The variables on the left of the equal sign may be subscripted variables (indexed arrays). Likewise, the expressions on the right of the equal sign may contain subscripted variables. Both sides of the LET statement must be of the same type.

2.6 Arithmetic Operators

The arithmetic operators are used to combine numeric constants and variables into expressions. The following table illustrates these operators:

SYMBOL	OPERATION	EXAMPLE
-	Negation	-A
+	Addition	A+B
-	Subtraction	A-B
*	Multiplication	A*B
/	Division	A/B
^	Exponentiation	A^2

FIGURE 2.6.1 Arithmetic Operators

The unary minus negates the value following it. The four arithmetic operators perform the standard arithmetic functions. And the exponentiation symbol raises the first value to the power of the second.

2.7 String Operator

String variables, constants and expressions can be concatenated (joined together into a single string) by using the the '&' operation. When this symbol is placed between two strings, they are concatenated. In the following example, S1$ is set equal to "ABCDEFGHIJ":

```
S2$ = "EF" & "G"
S1$ ="ABCD" & S2$ & "HIJ"
```

2.8 Relational Operators

Relational operators are used to compare two expressions of the same type (numeric or string). Relational expressions can be employed within control flow statements. They may also be used to evaluate to the numeric values of -1 for true and 0 for false. These values (-1 and 0) may then be used within arithmetic expressions or they may be printed. The relational operators all have the same precedence. The following table lists them:

SYMBOL	OPERATION	EXAMPLE
=	equal to	A = B
<	less than	A < B
>	greater than	A > B
<= or =<	less or equal	A <= B
>= or =>	greater or equal	A >= B
<> or ><	not equal	A <> B

FIGURE 2.8.1 Relational Operators

The following example shows how relational expressions can be evaluated to numeric quantities:

```
DISPLAY 7+8=15; 2 = 2.0/.1; 100 >= 1

-1 0 -1
```

Comparisons between strings are based on the ASCII value of their characters. For example "A BIRD" < "A bird" because the ASCII value for lower case b is greater then the value for upper case B. Also "$ZEBRA" <= "AARDVARK" because the code for $ is less then (or equal to) the code for A. If strings are identical except

that one string has addtional characters, then the longer string is greater: "COW" < "COWBOY".

2.9 Logical Operators

Logical operators are used within expressions to create results which have the values of TRUE or FALSE. The three logical operators are NOT, AND and OR. The following truth table illustrates the actions of these operators:

X	Y	NOT X	X AND Y	X OR Y
F	F	T	F	F
F	T	T	F	T
T	F	F	F	T
T	T	F	T	T

FIGURE 2.9.1 Logical Operators

The NOT operator yields the value which is logically opposite the value of the argument.

The AND operator produces a TRUE if and only if both arguments are true.

The OR operator produces a FALSE if and only if both arguments are false.

The precedence of these operators is: NOT, AND, OR. This precedence may be overridden by using parentheses. The following examples illustrate the use of the logical operators:

```
A=1
B=2
C=3
IF NOT A > 0 THEN DISPLAY "TRUE" ELSE DISPLAY "FALSE"
IF A < B AND C < B THEN DISPLAY "TRUE" ELSE DISPLAY "FALSE"
IF A < B OR  C < B THEN DISPLAY "TRUE" ELSE DISPLAY "FALSE"
IF NOT A<B AND C<B THEN DISPLAY "TRUE" ELSE DISPLAY "FALSE"
IF NOT (A<B AND C<B) THEN DISPLAY "TRUE" ELSE DISPLAY "FALSE"

FALSE
FALSE
TRUE
FALSE
TRUE
```

These operators can also be used to manipulate integer values. The meaning of the logical operations on arithmetic bits is given by replacing every F with a 0, and

every T with a 1 in FIGURE 2.9.1. The NOT of an integer is equal to the NOT of each individual bit within it (the one's complement value). Likewise, the AND or OR of two integers is the bitwise AND or OR operation performed on them. For example:

```
DISPLAY NOT 0
DISPLAY NOT -2
DISPLAY 1981 AND 255
DISPLAY 1981 OR  255

-1
1
189
2047
```

Negative one is the bitwise complement of zero. Similarly, one is the complement of negative two. The number 255 is a byte of all ones. The AND of 255 and 1981 represents the lower byte of 1981. The OR of 255 and 1981 is the upper byte of 1981 and a lower byte of 255.

Floating point numbers may be used with the logical operators. However, they are converted to integer form before they are operated upon.

2.10 Precedence of Operators

The arithmetic and logical operations discussed in this chapter are evaluated according to the following priorities:

PRIORITY	OPERATION	SYMBOL
1	Exponentiation	^
2	Unary Minus	-
3	Multiplication and Division	*,/
4	Addition and Subtraction	+,-
5	Relational Operators	=,<,>,<=,>=,<>
6	Logical NOT	NOT
7	Logical AND	AND
8	Logical OR	OR

FIGURE 2.10.1 Precedence of Operators

Using these priorities, expressions are evaluated from left to right. A portion of an expression may be placed inside parentheses. In this case, it will be evaluated separately before being combined with the rest of the expression. Within the parentheses, the same order of precedence is held. Parentheses may be nested, and

the most deeply nested portion of the expression is evaluated first.

2.11 Evaluation of Expressions

All expressions evaluate to either numeric or string values. (If the result is a logical value, it is actually stored as a numeric quantity.) The last operator evaluated, based upon the operator precedence discussed in Section 2.10, determines what the expression type is.

2.11.1 Arithmetic Expressions

An arithmetic expression is one in which the last operator evaluated is an arithmetic operation. Any combination of numeric variables, subscripted numeric variables, numeric constants, or numeric functions along with arithmetic, logical and relational operators may be combined together to form an arithmetic expression. The only restriction is that two variables, constants, or operators may not appear in direct succession. (A unary minus may appear, however, after another operator; for example A*-B.) The following are correct arithmetic expressions:

```
A
A+B*C
1.2*A/B
-A/-B
A^B^C
(A AND B)*-1
(A >= B)+1
```

String functions and mathematical functions, discussed in later chapters, may also appear within numeric expressions.

2.11.2 Logical Expressions

There is no short-circuiting done during the evaluation of logical expressions. In the expression:

```
A = B AND C > D
```

both clauses, (A=B) and (C>D), are evaluated even if the first one evaluated is found to be false. Then the AND operation is performed.

2.11.3 String Expressions

String constants, string variables, and string arrays may be combined with the string operators described in Sections 2.7, 2.8, and Chapter 5 to form string expressions. The results of these expressions are string values.

2.11.4 Relational Expressions

Constants and variables may be combined with arithmetic and logical operators to form relational expressions. The only requirement is that the last operation performed must be a relational operation. Relational operations are often used within control flow constructs such as IF THEN ELSE statements.

CHAPTER 3

I/O STATEMENTS

3.1 The PRINT and DISPLAY Statements

String or numeric expressions may be output using the DISPLAY or PRINT statements. Usually the expressions to be output are constants or single variables. The formatting of the output can be controlled with these statements.

The PRINT statement directs output to the printer unless another unit is specified. If there is no printer, the PRINT statement directs its output to the console. The DISPLAY statement directs output to the console (video terminal). If a PRINT statement is re-directed to the console, it acts on all of the options described in this chapter. If not, it ignores the following options: ERASE ALL, AT, SIZE and BELL. Except for these differences, the two statements are identical. The description in this chapter of the DISPLAY statement and its options applies equally to the PRINT statement.

The DISPLAY statement may be used with or without an expression following it:

```
DISPLAY
DISPLAY expression
```

If no expression is indicated, then a carriage return is output. Otherwise, the value of the (numeric or string) expression is output, followed by a carriage return. When string values are displayed, no automatic formatting is done. Numeric values, however, are displayed with a leading character and a trailing blank. The leading character is a space if the value is positive, a minus sign is the value is negative. The trailing blank is used to separate numbers which are directly adjacent on the same line. The following are examples of the DISPLAY statement using expressions:

```
DISPLAY "Roses are red, Violets are blue"
DISPLAY 2+2

Roses are red, Violets are blue
 4
```

The DISPLAY statement may be followed by a list of expressions:

```
DISPLAY list
```

A list is several expressions, separated by commas, semicolons, or apostrophes. These expressions may be mixtures of string and numeric types. The commas, semicolons, and apostrophes are known as data separators. The effect of each data separator is different. But ending a list with any of them causes the cursor to stay on the current line after the DISPLAY statement is executed. In this way, it is possible to use more than one DISPLAY statement to print characters on a single line.

An output line is divided into display zones which are 16 characters wide. The comma causes the cursor to advance to the next display zone. If the cursor is currently in the last zone, then it advances to the first zone on the next line. Data is left-justified within each zone. The following is an example of the use of the comma as a data separator:

```
DISPLAY 1,2,3
DISPLAY "DOG","CAT","BIRD"

 1               2               3
DOG             CAT             BIRD
```

This same effect can be achieved by using separate DISPLAY statements separated by commas:

```
DISPLAY 1,
DISPLAY 2,
DISPLAY 3
DISPLAY "DOG",
DISPLAY "CAT",
DISPLAY "BIRD"

 1               2               3
DOG             CAT             BIRD
```

Using a semicolon between expressions causes NO separation between them:

```
DISPLAY 1;2;3
DISPLAY "DOG";"CAT";"BIRD"

 1  2  3
DOGCATBIRD
```

The spaces between the 1, 2, and 3 represent the leading and trailing blanks which always accompany numeric values.

Using an apostrophe between expressions causes a comma to be inserted between them:

```
DISPLAY 1'2'3
DISPLAY "DOG"'"CAT"'"BIRD"

 1 , 2 , 3
DOG,CAT,BIRD
```

3.2 Options available with the PRINT and DISPLAY statements

This section discusses the several options which can be used with the DISPLAY and PRINT statements. These options may be combined to fully format the output as desired. The list of options follow the DISPLAY or PRINT command. A colon is placed after the last option, and the expressions to be output are then specified. If more than one of the options ERASE ALL, AT, SIZE, or BELL are used, they must be in the order shown in this sentence.

3.2.1 The ERASE ALL Option

The ERASE ALL option causes the screen to be cleared before values are displayed. The following example illustrates the syntax for this option:

```
DISPLAY ERASE ALL: "DOGS"," CATS"," and lots of BIRDS"
```

3.2.2 The AT Option

The AT option can be used to indicate a starting line and column number for the display to appear on the screen. Column and line numbers start at 1. The format for this option is:

```
AT (line_number,column_number)
```

The default line number is 24 (the bottom line on the screen). The default column number is 1. If a DISPLAY statement which uses an AT option is followed by a DISPLAY statement which does not, the second statement starts in the default position regardless of where the first was positioned. The following example illustrates the use of the AT option:

```
COL=10
DISPLAY AT (12,COL): "Where am I?"
```

3.2.3 The SIZE Option

The SIZE option can be used to specify the maximum number of characters to be output by a DISPLAY statement. The format for the SIZE option is:

```
SIZE (n)
```

If this option is not used, the default size is large enough to hold all of the characters to be output, plus enough extra blank spaces to fill the end of the last line onto which the DISPLAY statement is writing. When this option is used, the line will be cleared, after the output is performed, only as far as the indicated size. If it is desired to leave portions of a line intact when displaying to the same

line, this can be done with the SIZE option. The semicolon data separator should be placed at the end of the display list so that no clearing will be done beyond the end of the last character being displayed. Strings which have a length greater than the specified size will be truncated on the right. If a negative size is given, its absolute value will be used. The following example illustrates the use of the SIZE option:

```
DISPLAY SIZE(38): "There are 38 characters in this string"
```

3.2.4 The BELL Option

The BELL option causes the terminal bell to ring when the display statement is executed. The following are examples:

```
DISPLAY BELL: "There were bells on the hills"
DISPLAY AT(10,10) BELL: "But I never heard them ringing"
```

3.2.5 The USING Option

The USING option controls the output of a DISPLAY statement. (The USING option may also be employed within the ASSIGN statement in conjunction with Virtual Arrays, see Section 7.1.7, and within the PRINT statement in conjunction with file I/O, see Section 7.2.1.) It has the following format:

```
USING line_number
USING string_expression
```

The line number is the number of a line containing an IMAGE statement (see Section 3.2.6). The string expression contains the elements of an IMAGE statement.

If a DISPLAY statement employs a USING statement, then the list of expressions to be displayed must use commas as data separators. Also, the only data separator that may terminate this list is a semicolon. The following illustrates the use of the USING option:

```
      NUM=999.999
      FORMAT$="###.##"
10    IMAGE ###.##
      DISPLAY USING "###.##":NUM
      DISPLAY USING FORMAT$:NUM
      DISPLAY USING 10:NUM

      999.99
      999.99
      999.99
```

All of the above statements produce the same output. If a DISPLAY statement list contains more expressions than the corresponding IMAGE statement contains formats, a new line is begun. This new line is formatted according to the same IMAGE statement. If there are fewer expressions than IMAGE formats, the DISPLAY terminates after the last expression.

3.2.6 The IMAGE Statement

The IMAGE statement is referred to by line number within the USING clause of a DISPLAY or PRINT statement. It provides a format for the expressions to be output. It has the format:

IMAGE string_constant

The quotes around the string constant are optional. Text may be inserted into the string constant and this text will appear in exactly the same position in the actual output. Text consists of all characters which are not format control characters.

3.2.6.1 Format Control Characters

There are nine format control characters: #, ^, -, +, ., <>, ,, $, and *.

The # (number sign) indicates the place of a data character.

The ^ (exponent sign) indicates how many places an exponent should fill. If there are more places indicated then the actual exponent has, leading zeros are inserted. Either four or five exonent signs should be used. If less than four exponent signs are used, they will be printed as a literal string rather than used to indicate an exponent field.

The - (minus sign) specifies the position of the minus sign if the value is negative. If the value is positive, this position will be left blank. The minus sign may be placed before or after the value.

The + (plus sign) may be placed to the left of a numeric field. It indicates that positive numbers are to be displayed with a plus sign preceding them. Negative numbers are displayed with a minus sign as usual.

The . (decimal point) is used to indicate the position of the decimal place.

The <> (angle brackets) are used to enclose numeric IMAGE fields if it is desired to have negative numbers appear within angle brackets. Positive numbers will appear without the brackets.

The , (comma) will produce a comma at the specified position within a numeric value.

The $ (dollar sign) will cause a dollar sign to appear at the beginning of the indicated field. A $$ (double dollar sign) allows the dollar sign to float (otherwise it is left justified).

Two ** (asterisks) produce asterisk fill wherever the numeric value does not fill the field. This is used in protecting checks.

The following is an example of the use of these format characters:

```
      A=999.999
      B=88.8888
      S$="OCEAN"
10    IMAGE ##### BLUE <###.##>
      DISPLAY USING 10:A,B
      DISPLAY USING 10:S$,-B

        1000 BLUE    88.89
      OCEAN BLUE   <88.89>
```

3.2.6.2 Fields Within IMAGE Definitions

An integer field of an IMAGE definition or string has no decimal point. It may have a sign. If the value overflows the field, asterisks will be produced instead of the value. The integer is right-justified within the field, and is rounded. The following example illustrates integer fields:

```
10    IMAGE "#### ##### ###"
      I=999 :: J=-88 :: K=7777
      DISPLAY USING 10: I,J,K

       999  -88  ***
```

A decimal field consists of a string of number signs and may have a plus/minus sign. A decimal point may appear within it, just before it, or just after it. The value is rounded according to the quantity of number signs which follow the decimal point in the IMAGE format. The number is right-justified within the field. The decimal point is placed in the position indicated in the field definition. If the number overflows, asterisks are displayed instead of the value. The following example illustrates decimal fields:

```
10    IMAGE "#### ##### ###"
      I=111.11 :: J=-88.888 :: K=7777.7777
      DISPLAY USING 10: I,J,K

       111  -89  ***
```

An exponent field is a series of four or five exponent signs (^) which reserve space for the exponent. The number is rounded similarly to decimal fields. A leftmost plus/minus sign reserves space for the appropriate sign. If the minus sign is used, a blank will appear if the value is positive. There must be at least one character (#, + or -) to the left of the decimal point if the number to be displayed is negative. The following example illustrates exponent fields:

```
20      IMAGE "#.#####^^^^ ##.###^^^^ ###.^^^^ #.##^^^^"
        A=111.111 :: B=-66.666 :: C=55.5 :: D=-.077
        DISPLAY USING 20:A,B,C,D

         .11111E+03 -6.667E+01  56.E+00  -.78E-01
```

A string field may be indicated by any sequence of control characters. If the string is shorter than the indicated field, blank spaces will be padded on the right. If the string exceeds the specified length, it will be truncated on the right.

Fields consisting of characters other than the control characters are taken as text to be literally inserted into the displayed output.

3.2.6.3 The PUNCTUATION Statement

The PUNCTUATION statement can be used to alter the monetary symbols for currency ($), digit separators (,) and decimal point (.). This statement takes the following form:

```
PUNCTUATION string_expression
```

The first character in the string expression is used for the currency symbol. The second is used for the decimal point. The third character is used for the digit separator symbol. The default values for these are the same as they would be if the following statement was executed:

```
PUNCTUATION "$.,"
```

The following example demonstrates the use of the PUNCTUATION statement:

```
10      IMAGE $$###,###.##
        AMOUNT=999999.25
        DISPLAY USING 10:AMOUNT
        PUNCTUATION "L,."
        DISPLAY USING 10:AMOUNT

        $999,999.25
        L999.999,25
```

3.3 The INPUT Statement

The INPUT statement accepts values typed in from the keyboard during program execution. The basic form of this statement is:

```
INPUT variable
```

A question mark, followed by a blank space, appears when this statement is executed. When a value is entered, followed by a <return>, the variable is assigned accordingly. If it is a string variable, the input will be interpreted as a string. If it is a numeric variable, the input must represent a correct numeric value. Leading and trailing blanks are removed from string variables.

Several variables may be included within the INPUT statement if they are separated by commas. The keyboard input must be made on a single line and the input values must be separated by commas. All the variables must be within a single INPUT statement for each line input form the keyboard. This last constraint does not apply when input is being done from files.

A quoted string followed by a colon may precede the variable list. The string will be used as a prompt to replace the question mark. If no prompt and no question mark are desired, a null string ("") may be used. The following are examples of the INPUT statement:

```
INPUT RATE
INPUT "":HEIGHT,WIDTH,NAME(1)
INPUT "Type a character string:":STRING1$
```

3.3.1 The AT Option With the INPUT statement

The AT option may be used with the INPUT statement in a manner similar the DISPLAY statement. The cursor is positioned according to the AT clause specifications. The following are examples of the AT clause:

```
INPUT AT(10,10):PAY
INPUT AT(10,10)"Enter Pay":PAY
```

3.3.2 The SIZE Option With the INPUT statement

The SIZE option can be used to specify the maximum number of characters that may be input. If the number specified is positive, the line will be cleared before a prompt for input is made. If that number is negative the line will not be cleared. The bell will sound if more characters are entered than the SIZE clause allows. The default size is the remainder of the line after the input prompt. If a size is

specified, it does not include the length of an input prompt if one is issued. The following are examples of the use of the SIZE option with the INPUT statement:

```
INPUT AT (10,18) SIZE(2),"What year?": YEAR
INPUT SIZE(-3):S$
```

If the input exceeds the specified size, the bell will sound for each extra character typed until a <space> or <return> is input.

3.3.3 The BELL Option With the INPUT statement

When the BELL option is used with the INPUT statement, the bell will ring, prompting the user to input. The following is an example of this:

```
INPUT AT(10,20) BELL,"I hear bells, do you?": S$
```

3.4 The ACCEPT Statement

The ACCEPT statement is used like the INPUT statement, except that it can take only one variable. The ACCEPT statement reads the entire line as input and does not edit out commas or quotes. Since commas are used as data separators for the INPUT statement, the ACCEPT statement is useful because a comma can be a part of an input string. The ACCEPT statement, therefore, is most useful when reading into a string variable. The INPUT statement options described above may also be used with the ACCEPT statement. The following are examples of the ACCEPT statement:

```
ACCEPT S$
ACCEPT "What Company? ": CO$
ACCEPT AT(10,10) SIZE(2): DAY_OF_MONTH
```

3.5 The DATA Statement

The DATA statement defines values that will be used as data within a program. These values may be numeric or string constants. Quotes may optionally be used to enclose string data. Strings must be enclosed in quotes if commas are contained within them. Otherwise, commas are interpreted as data separators. Also, leading and trailing blanks will be removed from strings which are not within quotes. The following is the DATA statement format:

```
DATA list
```

The list is one or more constants separated by commas. These constants may be numeric or string types.

Several DATA statements may appear within a program. They may be placed anywhere in the program source text and need not be grouped together. As one DATA statement is exhausted, the next one in the file will be used. The following are examples of the DATA statement:

```
DATA 20,40,60,80
DATA 100,120,140,160
      .
      .
      .
DATA "CALIFORNIA"
DATA "TEXAS"
```

3.6 The READ Statement

The READ statement uses the values specified in DATA statements. It assigns these values to variables which are listed in the READ statement. The READ statement variables may be numeric or string and they may be subscripted or unsubscripted. The DATA statements will be used in the order that they appear in the source text. The specified variables and the corresponding data values must be of the same type and have the same range. The READ statement has the following form:

```
READ list
```

The list is one or more variables separated by commas.

If a READ statement is encountered and no corresponding DATA statement has been declared, or if all the DATA statements have been exhausted, then an error will occur. The following illustrates the use of the READ statement:

```
READ I,J
DISPLAY I;J;
READ I,J
DISPLAY I;J
DATA 2,4,6,8

 2  4  6  8
```

3.7 The RESTORE Statement

During program execution an internal data pointer is kept. This pointer indicates the next DATA statement value to be read. The RESTORE statement resets this pointer to the first DATA statement in the program. Alternatively, the line number of a particular DATA statement may be specified, and the RESTORE statement will reset the data pointer to that statement. After the RESTORE statement is executed, the next READ statement will take its input from where the reset data pointer indicates. The RESTORE statement takes the following forms:

```
RESTORE
RESTORE line_number
```

If a line number is indicated and that line does not contain a DATA statement, then the next line which does contain a DATA statement is used. If there is no DATA statement on the indicated line or following it, then an error will occur at the next READ statement. The following illustrates the use of the RESTORE statement:

```
      DATA 1,2
20    DATA 3,4
30    DATA 5,6
      READ I,J,K,L
      DISPLAY I;J;K;L
      RESTORE
      READ I,J,K,L
      DISPLAY I;J;K;L
      RESTORE 20
      READ I,J,K,L
      DISPLAY I;J;K;L

 1  2  3  4
 1  2  3  4
 3  4  5  6
```

CHAPTER 4

CONTROL FLOW STATEMENTS

4.1 The GOTO Statement

The GOTO statement unconditionally transfers control to a specified line number. It has the following format:

```
GOTO line_number
GO TO line_numer
```

The following sample program shows the use of the GOTO statement:

```
      I=1
10    DISPLAY I
      I=I*2
      GOTO 10

       1  2  4  8  16  ...
```

4.2 The ON-GOTO Statement

The ON-GOTO statement allows a multiple switch mechanism for control flow. This statement has the format:

```
ON expression GOTO line_num_1, line_num_2, line_num_3 ...
```

The expression is any valid numeric expression. If necessary it will be rounded to an integer. If the expression evaluates to 1, control is transfered to the first line number. If the expression evaluates to 2, control is transfered to the second, etc. If the expression is less than 1 or greater than the number of listed line numbers, an error will result. The following example illustrates the use of the ON-GOTO statement:

```
        I=0
10      I=I+1
        ON I GOTO 20,30,40
20      DISPLAY "AT LINE 20
        GOTO 10
30      DISPLAY "AT LINE 30
        GOTO 10
40      DISPLAY "AT LINE 40
END

        AT LINE 20
        AT LINE 30
        AT LINE 40
```

4.3 The IF-THEN-ELSE Statement

The IF-THEN-ELSE statement provides conditional transfer of control flow based on the value of a relational expression. It has the following forms:

```
IF condition THEN action
IF condition THEN action_a ELSE action_b
```

The condition is a relational expression. If the expression evaluates to true, the THEN clause is executed. Otherwise, the ELSE clause is executed (if there is no ELSE clause, the next statement in the program is executed). The action is either a single executable statement, or a line number to which control will be transfered. The entire statement must fit on one line (or several lines joined together with comment delimiters, see Chapter 1). The following example demonstrates the use of this statement:

```
        S1$="ABC"
        S2$="123"
        IF S1$=S2$ THEN 10 ELSE 20
10      DISPLAY "WHAT ??"
        STOP
20      DISPLAY "GOOD"
        END

        GOOD
```

The IF clause condition may be a numeric expression. In this case the resulting value is taken to be false if its least significant bit is zero, and true otherwise.

```
A=10
IF A=10 THEN GOOD=1 ELSE GOOD=0
IF GOOD THEN DISPLAY "GOOD" ELSE DISPLAY "BAD"

GOOD
```

4.4 The FOR-TO-STEP and NEXT Statements

These statements are used to create programming loops. They have the following formats:

```
FOR variable = init_val TO final_val
FOR variable = init_val TO final_val STEP increment
NEXT
NEXT variable
```

The variable is any subscripted or unsubscripted numeric variable. If it is a subscripted variable such as A(10,J), its actual location is confirmed the first time the loop is executed and does not change within the loop. Init_val, final_val and increment are any valid numeric expressions. They are also bound at the first execution of the loop, and do not change. When the FOR statement is first executed, the variable is assigned the value of init_val. When the NEXT statement encountered, the value of increment is added to it. If no increment is specified, 1 is added. If the value of the increment is positive and the new value of variable does not exceed final_val, the loop is re-executed. Likewise, if the value of the increment is negative and the new value of variable is not less than final_val, the loop is performed again.

The loop consists of those statements which lie between the FOR statement and the NEXT statement. It is possible that the loop will never be executed if the increment is positive and final_val is less than init_val, or if the increment is negative and final_val is greater than init_val.

The NEXT statement may be followed by the loop variable. If this is the case, that variable must match the variable specified in the preceding FOR statement. If the loop variable is an array element, the NEXT statement should only specify the array name.

The following demonstrates the use of these statements:

```
FOR J=0 TO 10 STEP 2
  DISPLAY " J=";
  DISPLAY J;
NEXT
```

```
 J= 0   J= 2   J= 4   J= 6   J= 8   J= 10
```

FOR statement loops may be nested as follows:

```
FOR I=1 TO 10
    .
    .
    .
  FOR J(2,3,4)=A TO B STEP C
      .
      .
  NEXT J
    .
    .
    .
NEXT I
```

4.5 The GOSUB and RETURN Statements

Basic programs may have procedure blocks within them. A procedure block is a group of statements which are called by the GOSUB statement. When a RETURN statement is encountered the block is exited and execution is continued at the first statement after the GOSUB call. These statements have the following format:

```
GOSUB line_number
    .
    .
    .
RETURN
```

The line_number indicates the start of the procedure block. When the RETURN is encountered the block is exited. The following example illustrates the use of these statements:

```
      I=10
      J=20
      GOSUB 100
      I=100
      J=200
      GOSUB 100
          .
          .
          .
100   DISPLAY I;J;
      DISPLAY I+J
      RETURN

       10   20   30
       100   200   300
```

Procedure blocks may be nested in the following fashion:

```
      I=10
      GOSUB 100
      I=20
      GOSUB 100
          .
          .
          .
100   IF I=20 THEN GOSUB 200 ELSE DISPLAY "In block 100"
      DISPLAY "This is the second statement in block 100"
      RETURN

200   DISPLAY "In block 200"
      RETURN

      In block 100
      This is the second statement in block 100
      In block 200
      This is the second statement in block 100
```

4.6 The ON-GOSUB Statement

The GOSUB statement has a computed format similar to the computed GOTO statement:

```
ON expression GOSUB line_1, line_2, ...
```

If the expression is equal to 1, control is transfered to the first line indicated. If

the expression is equal to 2, the second line indicated is chosen, etc. Like the computed GOTO statement, an error will result if the expression is less than one or greater than the number of listed line numbers. The following example illustrates the use of the computed GOSUB statement:

```
      I=1
10    ON I GOSUB 100,200,300
      I=I+1
      IF I <= 3 THEN 10
          .
          .
          .
100   DISPLAY "Block 100"
      RETURN
200   DISPLAY "Block 200"
      RETURN
300   DISPLAY "Block 300"
      RETURN

      Block 100
      Block 200
      Block 300
```

4.7 The END and STOP Statements

The END statement is used to indicate that the end of a program has been reached. It must be the last statement in a program. It has the format:

```
      END
```

The STOP statement causes execution to terminate. There may be more than one STOP statement in a program. It has the form:

```
      STOP
```

The following illustrates the use of these statements:

```
      ACCEPT "Enter a number": I
      IF I > 0 THEN 10
      DISPLAY I
      STOP
10    I=-I
      DISPLAY I
      END
```

CHAPTER 5

STANDARD FUNCTIONS

5.1 The Numeric Functions

The numeric functions take as an argument a numeric constant, variable, or expression. These functions may be used within assignment statements, PRINT or DISPLAY statements, ON statements, and function definitions.

5.1.1 The ABS Function

The ABS function returns the absolute value of the argument. A nonnegative argument will be returned unaltered. The following is an example of the ABS function:

```
I=2
J=-3
DISPLAY ABS(I); ABS(J)

 2  3
```

5.1.2 The SIN Function

The SIN function returns the sine of the argument passed in radians. In order to convert an angle from degrees to radians, multiply the number of degrees by PI/180. The following example illustrates the SIN function:

```
I=25.0
DISPLAY SIN(I)

-.13235175
```

5.1.3 The COS Function

The COS function returns the cosine of the argument passed in radians. In order to convert an angle from degrees to radians, multiply the number of degrees by PI/180. The following example illustrates the COS function:

```
I=25.0
DISPLAY COS(I)

.99120292
```

5.1.4 The TAN Function

The TAN function returns the tangent of the argument passed in radians. In order to convert an angle from degrees to radians, multiply the number of degrees by PI/180. The following example illustrates the TAN function:

```
I=25.0
DISPLAY TAN(I)

-.1335264
```

5.1.5 The ATN Function

The ATN (arctangent) function returns the angle in radians, which has the tangent equal to the argument passed. If degrees are desired, multiplying the output of the ATN function by 180/PI performs the conversion. The following example illustrates the ATN function:

```
I=25.0
DISPLAY ATN(I)

1.5308176
```

5.1.6 The EXP Function

The EXP function yields the value of e, the base of natural logarithms, raised to the power of the argument passed. The following example illustrates the EXP function:

```
I=25.0
DISPLAY EXP(I)

72005171000.0
```

5.1.7 The LOG Function

The LOG function yields the natural logarithm (base e) of the argument passed. The following example illustrates this function:

```
I=25.0
DISPLAY LOG(I)

3.2188754
```

5.1.8 The INT Function

The INT function returns the largest integer which is not greater than the argument:

```
DISPLAY INT(25.9); INT(-3.2)

 25 -4
```

5.1.9 The SGN Function

The SGN function returns 0 if the argument is zero, 1 if the argument is positive, and -1 if the argument is negative:

```
I=0
J=749
K=-1024
DISPLAY SGN(I); SGN(J); SGN(K)

 0  1 -1
```

5.1.10 The SQR Function

The SQR function yields the square root of the value passed. If the argument is negative, an error results. The following example illustrates the use of the SQR function:

```
I=25
DISPLAY SQR(I)

5
```

5.1.11 The RND Function and RANDOMIZE Statement

The RND function produces evenly distributed pseudo-random numbers which fall in the range X: 0 <= X < 1. It has the format:

```
RND
```

The RANDOMIZE statement can be used to specify a "seed value" which will generate a new sequence of numbers. It has the following format:

```
RANDOMIZE numeric_expression
RANDOMIZE
```

The same sequence of values will be produced by the RND function in different programs whenever no RANDOMIZE statement is used, or whenever there are two occurrences of the RANDOMIZE statement with the same seed value. If the RANDOMIZE statement is used without specifying a value, the seed value will be taken from the real-time clock. If there is no clock, the user will be prompted to enter a seed value. This will provide for an uncontrolled sequence of numbers.

The following example illustrates the use of these statements:

```
RANDOMIZE (.12345678)
FOR I=1 TO 5
  DISPLAY RND
NEXT I

.6213
.97039985
.41409874
.51999116
.2612381
```

5.2 String Functions

The string functions are used in conjunction with string variables. These functions may be used within assignment statements, PRINT or DISPLAY statements, ON statements, and function definitions.

5.2.1 The ASC Function

The ASC function returns the decimal value of the ASCII code for the first character in the string argument. It has the format:

```
ASC (string)
```

The string may be any valid string expression. The following example illustrates the use of the ASC function:

```
S$="&"
DISPLAY ASC(S$)

 38
```

5.2.2 The BREAK Function

The BREAK function finds the first character in a string that appears in a second string. It has the format:

BREAK (string_1, string_2)

Strings 1 and 2 are any valid string expressions. This function compares the first character in string_1 to all the characters in string_2. If there is no match, it compares the second character in string_1 to all the characters in string_2, etc. It returns the number of characters in string_1 which did not match any character in string_2 before a matching character was found. If no match was found, it returns the total number of characters in string_1. The following example illustrates this function:

```
S1$="ABC"
S2$="COW"
S3$="XXXX1"
DISPLAY BREAK (S1$,S2$)
DISPLAY BREAK (S2$,S1$)
DISPLAY BREAK (S1$,S3$)
DISPLAY BREAK (S3$,S1$)

 2
 0
 3
 5
```

5.2.3 The SPAN Function

The SPAN function compares the characters in one string with the characters in a second string until a character in the first string is not found in the second string. It has the format:

SPAN (string_1, string_2)

String_1 and string_2 are any valid string expressions. Consecutive characters of string_1 are compared to characters in string_2. When a character in string_1 is found which is not in string_2, SPAN returns the number of characters that did match. The following example shows the use of the SPAN function:

```
S1$="$$$ Hi there"
S2$="$"
DISPLAY SPAN(S1$,S2$)

 3
```

5.2.4 The LEN Function

The LEN function yields the number of characters in the string passed. It has the following format:

```
LEN (string)
```

The string is any valid string expression. The following example shows the use of the LEN function:

```
S1$="ABC"
S2$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
DISPLAY LEN (S1$)
DISPLAY LEN (S2$)

 3
 26
```

5.2.5 The NUMERIC Function

The NUMERIC function will determine whether or not the string passed represents a valid number. A -1 will be returned if it does, and a 0 will be returned if it does not. If a string represents a valid number, it may be passed to the VAL function (see Section 5.2.6). The following example illustrates the use of the NUMERIC function:

```
DISPLAY NUMERIC ("1234"); NUMERIC ("12ABC")

-1  0
```

5.2.6 The VAL Function

The VAL function returns the numeric value of the string argument. Leading and trailing blanks are permitted. Any string expression which is accepted by the NUMERIC function (see Section 5.2.5) may be passed to the VAL function without error. The following example illustrates the use of the VAL function:

```
S$="1.234"
K=VAL(S$)
DISPLAY K+0

 1.234
```

5.2.7 The STR$ Function

The STR$ function provides a counterpart to the VAL function. The STR$ function is passed a numeric value and returns the corresponding string value. It has the format:

```
STR$(numeric_expression)
```

The string returned is identical to the numeric value as it would appear on the console, i.e. it is preceded by a blank space or a minus sign, etc. The following example illustrates the use of the STR$ function:

```
1=1.234
S$=STR$(1)
DISPLAY (S$)

 1.234
```

5.2.8 The POS Function

The POS function is used to determine the position of one string within another. It has the following format:

```
POS (string_1, string_2, start)
```

String_1 and string_2 are any valid strings, and start is a numeric value. This function returns the position of the first occurrence of string_2 within string_1. The search will begin at character position start within string_1. Start is rounded to an integer value if necessary. If string_2 is not found within string_1, a 0 will be returned. The following example illustrates the POS function:

```
DISPLAY POS("ROW ROW ROW YOUR BOAT", "ROW", 1);
DISPLAY POS("ROW ROW ROW YOUR BOAT", "ROW", 2);

 1  5
```

5.2.9 The RPT$ Function

The RPT$ function returns a string which is a specified number of repetitions of the argument string. It has this format:

```
RPT$ (string, numeric_expression)
```

The string is any valid string. The numeric expression may evaluate to any nonnegative number less than 256. The string returned is that number of repetitions

of the string passed. If the resulting string has a length greater than 255 an error will result. The following example shows the use of this function:

```
DISPLAY RPT$("Cats ",3)

Cats Cats Cats
```

5.2.10 The UPRC$ Function

The UPRC$ function changes all lowercase letters in the string passed to upper case letters. Nonalphabetic characters remain the same. The following example illustrates the use of this function:

```
S$="Once upon a time..."
DISPLAY UPRC$(S$)

ONCE UPON A TIME...
```

5.2.11 The CHR$ Function

The CHR$ function takes a numeric argument, the value of which must fall between 0 and 255 inclusive. It returns a single-character string whose ASCII value is equal to that number. This complements the ASC function (see Section 5.2.1). Special control characters within DISPLAY or PRINT statements can be generated with this function. The following example shows the use of this statement:

```
DISPLAY CHR$(65)

A
```

5.2.12 The SEG$ Function

The SEG$ function extracts a segment of a string. It has the following format:

```
SEG$(string, position, length)
```

The string is any valid string expression. Position and length are numeric expressions which will be rounded to integers if necessary. Starting at position characters into the string, length characters will be extracted by the SEG$ function. If position is less than or equal to zero an error will result. If position is greater than the length of the string, a null string will be returned. If length is less than 0, an error will result. If length is equal to 0, a null string will be returned. If length plus position are greater than the remaining portion of the string, all of the string will be extracted starting at position. The following is an example of the use of the SEG$ function:

```
S$="But don't you step on my Blue Suede Shoes..."
DISPLAY SEG$ (S$,26,16)

Blue Suede Shoes
```

5.3 Miscellaneous Functions

The remaining standard functions discussed in this section are used in the same way as numeric and string functions.

5.3.1 The DAT$ Function

The DAT$ function returns the month, day and year in the form:

```
month/day/year
```

Month, day, and year are two-digit numbers. The following example displays the date using this function:

```
DISPLAY DAT$

02/16/81
```

5.3.2 The FREESPACE Function

The FREESPACE function returns the number of bytes available in memory. It has the following format:

```
FREESPACE(0)
```

If there were 5000 bytes available, the following would occur:

```
DISPLAY FREESPACE(0)

 5000
```

5.3.3 The INKEY and INKEY$ Functions

The INKEY function always returns zero. The INKEY$ function reads and removes a character from the keyboard input buffer, and returns a string consisting of that character. These functions have the following formats:

```
INKEY(0)
INKEY$(0)
```

The following example shows the use of the INKEY$ function:

```
S$=INKEY$(0)
```

5.3.4 The EOF Function

The EOF function is used to determine whether or not the end of a file has been read. It has the following format:

```
EOF(X)
```

X is a numeric expression that evaluates to the file number which was assigned when the file was opened. (See Chapter 7 for further information about files.) A zero is returned by the EOF function if the last record of the file has not yet been read. A 1 is returned if the last record has been read. A 2 is returned if an attempt has been made to read beyond the end of the file. A 4 is returned if the specified file number is not in use.

5.3.5 The FTYPE Function

This function always returns 0. The type of a file is determined by the name associated with it. See Chapter 7 for further information on files.

5.3.6 The TAB Function

The TAB function advances the cursor or printer head to a specified position. It has the following form:

```
TAB(position)
```

Position is the column number where the next output will begin. The position may be a numeric constant, variable, or expression. This number will be rounded to an integer if necessary. It must be nonnegative, and the value actually used is this number modulus the output width of the device. If the cursor is already past the specified position, it will be advanced to that position on the next line. The following illustrates the use of this function:

```
DISPLAY TAB(10);CAT;
DISPLAY TAB(12);CAT;

          CAT
            CAT
```

5.3.7 The ERR Function

The ERR function returns an integer error number which indicates the last exception which took place. If there has been no error, a zero is returned. The following illustrates the use of the ERR function:

```
IF ERR > 0 THEN DISPLAY "ERROR ";ERR
```

5.3.8 The TIME$ Function

The TIME$ function returns a string which represents the current time based on the real time clock (if the computer is equipped with one). The following example shows the use of this function:

```
DISPLAY TIME$

11:24:10
```

This call to TIME$ yielded 24 minutes and 10 seconds after 11.

CHAPTER 6

USER-DEFINED FUNCTIONS AND SUBROUTINES

6.1 Functions

Functions are defined using the DEF and FNEND statements. A function consists of one or more statements which are executed each time the function is called. A function may have parameters passed to it, and it may have local variables. Functions may also reference variables which are global to them. In the body of the function, a value is assigned to the function name. This value will be evaluated when the function is called, and then returned as the value of the function. A function may call itself and other functions recursively. Subroutines may also be called by functions and call functions in the same way (see Section 6.2). Functions and subroutines may be nested to any depth.

6.1.1 The DEF Statement

The DEF statement is used to indicate a function definition. It has the following formats:

```
DEF func_name
DEF func_name (param_list)
DEF func_type func_name
DEF func_type func_name (param_list)
```

The func_type and param_list are optional and are described in the following paragraphs.

For single-statement functions, the DEF statement defines the entire function. For multiple-statement functions, the DEF statement defines the function name, type, and parameters. Function definitions are commonly placed at the beginning of a program. The following example shows the use of the DEF statement to define a single statement-function:

```
DEF A_FUNC(I,J,K)=I*J*K+200
```

A_FUNC is the name of this function. I, J, and K are parameters passed to it. The specified number of parameters must be passed each time the function is called. The value returned is the expression on the right side of the equal sign. The following is an example of a multiple statement function-definition:

```
DEF A_FUNC(I,J,K)
A_FUNC=I*J*K+200
FNEND
```

This function is equivalent to the one line function above. If the function name is assigned a value more than once in the body of the function definition, the last assignment before the execution of the FNEND statement determines the runtime function value.

Function parameters are always passed by value. (Subroutine parameters are passed by reference, see Section 6.2.) Any number of parameters may be passed to a function. If necessary, the comment delimiters (* and *) may be used to comment out the end of the line so that more than one line may be used to declare the parameters. The parameters may be subscripted variables. In this case, they are defined exactly as they would be within a DIM statement. The following illustrates the use of comments to extend parameter lists and subscripted variables within parameter lists:

```
DEF A_FUNC(A,B,C,D,E,F,G,H,I, (*
          *) J,K,L,M)
DEF A_FUNC( A(3,4), B$(5)*4 )
```

The function name may be any legal variable name which corresponds to the function type (numeric or string). If the function returns a numeric value, the function type may be optionally specified as INTEGER, REAL or DECIMAL. For example:

```
DEF INTEGER A_FUNC (I,J,K)=I*J*K+200
DEF REAL B_FUNC (I,J,K)=I*J*K+200
```

String functions may be defined to return a string that has a maximum length:

```
DEF MONTH$ (JULIAN)*3
```

6.1.2 The FNEND Statement

The FNEND statement indicates the end of a multiple statement function definition. This statement must be the last statement within the function body. A single statement function does not require an FNEND statement. The following example illustrates the use of this statement:

```
DEF F
   .
   .
   .
FNEND
```

6.1.3 Calling Functions

User-defined functions, like the standard functions, may appear any place an expression of the function type is permitted. The type and number of parameters specified in the function definition must be passed at the call. An array element or an entire array may be passed (or any other type of variable which matches the definition). Only the array name is specified when an entire array is passed. The following are examples of function calls:

```
I=FUNC1(1,J,K)
S$=FUNC2$( S1_ARRAY$(1,J), S2_ARRAY$, INT )
DISPLAY FUNC3
DEF FUNC4(1)=1*FUNC3
```

6.2 Subroutines

Subroutines are similar to functions. There are three differences between them. First, the subroutine name may not be assigned a value as in a function. Rather than being used within assignment statements like functions, subroutines are called using the CALL statement (see Section 6.3). Second, parameters are passed to subroutines by reference (not by value as they are to functions). This means that if an assignment is made to a parameter within a subroutine, the contents of the actual location of that variable within the caller will be altered. And third, a subroutine may not reference variables which are global to it (see Section 6.4). This includes other subroutines or functions.

Subroutines, like functions, may be nested to any depth. Subroutines may call themselves recursively. Subroutines may also have any number of parameters.

6.2.1 The SUB Statement

Subroutines are defined using the SUB statement. The SUB statement has the following formats:

```
SUB subr_name
SUB subr_name (param_list)
```

This statement is similar to the DEF statement for functions (see Section 6.1.1).

Like functions, this heading is followed by one or more statements. The parameter list follows the same rules as for functions.

6.2.2 The SUBEND and SUBEXIT Statements

Subroutines are terminated with the SUBEND statement in the same way that functions are terminated with the FNEND statement. But subroutines must always include the SUBEND statement, unlike one-line functions which do not need the FNEND statement. A subroutine may be specified on a single line by using double colons to separate statements. The SUBEND statement must be the last statement of subroutine.

There may be zero or more SUBEXIT statements within the subroutine body. When a SUBEXIT statement is encountered, the subroutine is exited in the same manner as it would be had the SUBEND statement been executed. The following examples illustrate the use of the SUBEND and SUBEXIT statements:

```
SUB S1 (A) :: A=1 :: SUBEND

SUB S2 (I,J)
IF I=J THEN 100
      .
      .
      .
SUBEXIT
100   .
      .
      .
SUBEND
```

6.3 The CALL Statement

The CALL statement is used to invoke a subroutine. It has the following formats:

```
CALL subroutine_name
CALL subroutine_name (param_list)
```

When the subroutine returns, execution resumes at the statement immediately following the CALL statement. The number and type of the parameters within the CALL statement must match the definition of the subroutine. The actual variables passed as parameters may be altered by the subroutine. The following examples demonstrate the CALL statement;

```
CALL SUB1
CALL SUB2 (I,J,K)
```

```
CALL SUB3 ( A(1,J), S_ARRAY$(,) )
```

6.4 Local Variables and Parameters

Parameters and local variables are valid only within the body of the function or subroutine. Numeric parameters may be defined to be INTEGER, REAL, or DECIMAL by using those statements when the parameters are declared. String parameters may be declared using the DIM statement if desired. Local variables are declared using the INTEGER, REAL, DECIMAL, and DIM statements on the lines which follow the function or subroutine heading. The following examples illustrate the declaration of function and subroutine parameters and local variables:

```
DEF FUNC1
  INTEGER A,B
  REAL R1,R2
  DIM S$*10
      .
      .
      .
SUB SUB1(INTEGER J,K, S_ARRAY$(2,3)*10, S$ )
  DIM C$*20
  INTEGER AN_ARRAY(2,3,4)
      .
      .
      .
```

In the above examples, FUNC1 is defined to have local variables A and B as integers, R1 and R2 as reals, and S$ as a ten-byte string. Subroutine SUB1 has parameters integer J, default type K (real unless otherwise specified), two-dimensional string array S_ARRAY$ and string S$. SUB1 also has local variables C$, a twenty-byte string, and AN_ARRAY, a three-dimensional integer array.

All local variables are cleared to zero, and local strings are set to null, each time the function or subroutine is invoked.

All variables within a subroutine are local to it. A subroutine may not access any variables external to it except through its parameter list.

A function may access variables in the standard block-structured manner. Any variables which are global to a function up to the subroutine (or program) definition may be accessed. If variables are not declared as parameters or local variables within functions, they will be assumed to be global.

6.5 Line Numbers and Data Lists Within Subroutines and Functions

Statement line numbers are local to functions and subroutines. This means that any GOTO, GOSUB, ON, RESTORE line_number or PRINT USING line_number statements will refer to line numbers within the function or subroutine only.

DATA lists and READ data statements are local to subroutines and functions. A RESTORE is executed automatically on entry to subroutines and functions.

6.6 The USES and LIBRARY Statements

BASIC programs may use separately compiled BASIC, Pascal, or FORTRAN units. These units contain functions and subroutines (or procedures) which may be called by a BASIC program. Units reside on disk, either in SYSTEM.LIBRARY or in some user-created file.

Units may have interface text which declares variables, functions, and subroutines (or procedures) to be externally recognized by the host program. This text is compiled when, during compilation of the host BASIC program, a USES statement is encountered. The USES statement indicates that one or more specified unit(s) within SYSTEM.LIBRARY are to be used by the BASIC program. It has the following format:

```
USES unit_name1, unit_name2, unit_name3, ...
```

If the unit(s) reside in a file other than SYSTEM.LIBRARY, that file may be specified with the LIBRARY statement. It has the format:

```
LIBRARY "file_name"
```

The default library is SYSTEM.LIBRARY. Once the LIBRARY statement has been executed, the indicated file will serve as the library where all units are to be found by all USES statements, until another LIBRARY statement is executed. The following is an example of these two statements:

```
USES PASCALIO

LIBRARY "MY.LIB.CODE"
USES MY_UNIT1, MY_UNIT2, MY_UNIT3
USES MY_UNIT4

LIBRARY "#5:ANOTHER.CODE"
USES ANOTHER_UNIT
```

In this example, PASCALIO will be found in SYSTEM.LIBRARY, MY_UNIT1 through

MY_UNIT4 will be found in MY.LIB.CODE on the prefixed disk, and ANOTHER_UNIT will be found in ANOTHER.CODE on the disk in drive #5.

In the II.0 version of the UCSD p-System, after compilation the unit must be linked into the codefile. This may be done by invoking the Linker or, if the unit resides within SYSTEM.LIBRARY, by simply R(UNning the program from the main system prompt which will automatically use the Linker. In the IV.0 version of the UCSD p-System, the unit must be connected to the host codefile using the Librarian, or if the unit resides in SYSTEM.LIBRARY, it will be called directly from there during program execution. More information on units, the Linker and the Librarian may be found in the UCSD Pascal Users' Manual.

6.7 Pascal Interface Text Restrictions

The BASIC compiler will parse the interface sections of Pascal units subject to some restrictions. First, only the following simple types (which corespond to BASIC types) are allowed:

```
INTEGER
REAL
STRING or STRING[ <size> ]
ARRAY [ <dimensions> ] OF <one of the preceding simple types>
ARRAY [ <dimensions> ] OF ARRAY[ <dimensions> ]  OF ...
```

No other types, and no user-defined types are permitted. A second restriction is that no constants are allowed within interface sections. For example, the following would not be correct:

```
ARRAY [LOW_INDEX, HIGH_INDEX] OF INTEGER;
```

Arrays such as this may be declared if the indices are ordinary integers.

Procedure and Function declarations are allowed, as long as the type of the parameters and the function type conform to these same restrictions.

The BASIC program should refer to any numeric Pascal variables which exceed eight characters in length by the first eight characters only. Alternatively, the Pascal program may be written so that no externally reconizable numeric variables exceed that length. If a Pascal string variable does not exceed eight characters in length, it should be refered to by BASIC with a dollar sign appended. If it does exceed that length, if should be referred to by BASIC as its first eight characters with a dollar sign appended.

6.8 The UNIT Statement

In order to create a BASIC unit, which is separately compiled and used by a host BASIC, Pascal, or FORTRAN program, the UNIT statement is used. This statement has the following format:

```
UNIT unit_name
```

The UNIT statement should be at the beginning of the text. Following this heading, an interface section may be declared using the INTEGER, REAL, DECIMAL, and DIM statements. Following this, Functions and Subroutines may be declared. These routines will be accessible to the host program. There should be no main program in the unit. The following is an example of a BASIC unit:

```
UNIT MY_UNIT
  INTEGER I,J
  DIM S$*20

DEF A_FUNC (A,B,C)
  IF A > 0 THEN GOTO 10
     .
     .
     .
10   FNEND

SUB A_SUB (PARAM$)
  IF S$=PARAM$ THEN I=2
     .
     .
     .
  SUBEND

END
```

In the above unit, integers I and J, and string S$ are able to be referenced from the host program. They are essentially global variables in the host. A_FUNC and A_SUB are accessible to the host program also. They are global routines within the host as if they had been declared, like any other function or procedure, at the beginning of the host.

All variable, function and subroutine names should be distinguishable by their first eight characters if the unit is to be used by a Pascal host program. This is because Pascal only distinguishes identifiers by their first eight characters. Also, no special characters may be used in the BASIC variable, function and subroutine names because Pascal allows only alpha-numeric characters within identifiers. The

single exception to this is the dollar sign required at the end of a string variable name.

The BASIC compiler will convert the externally recognizable BASIC text into a Pascal interface section so that the Pascal compiler may use the compiled BASIC unit.

CHAPTER 7

FILE I/O AND VIRTUAL ARRAYS

7.1 Opening and Closing Files

SofTech Microsystems BASIC allows the user to access disk files. The file must be created, or opened if it already exists, before it can be accessed. The file should be closed before the program terminates. If an error occurs during program execution, the files left open will remain open.

7.1.1 The OPEN Statement

The OPEN statement will either open an existing file or create a new one. Once the file is open, records within it may be accessed until the file is closed. The OPEN statement has the following format:

```
OPEN #file_num: "file_name", attributes
```

File_num is a numeric expression which has a positive value less than 256. This number will be associated with the file as long as it is open. File_num should not be assigned to any other file until this file is closed. File_name is a valid UCSD p-System file name. This name may include a unit number (such as #4:FILE.TEXT) or a unit name (such as DISK1:FILE.TEXT). The attributes are one or more of the file attributes which determine: File Access Mode, File Organization, File Length, File Format, Record Type, and Record Length. These are discussed in the following sections.

If any of the attributes in the following two lists are used, they must appear in the order shown:

```
SEQUENTIAL - DISPLAY - VARIABLE - File Access Modes
RELATIVE - INTERNAL - FIXED - File Access Modes
```

7.1.2 File Access Modes

The OUTPUT access mode indicates that the file to be opened is to be created as a new file. New records may be written to a file declared with this mode. This mode must be used if a new file is to be created. A device, such as a disk drive, may not be opened with this mode, because a device cannot be created. The following example will create FILE.TEXT on the disk in drive #4 and associate the file with file number 1:

```
OPEN #1: "#4:FILE.TEXT", OUTPUT
```

The INPUT access mode indicates that the file may be read from. If INPUT is the only attribute used, an attempt to write to the file will result in an error. If both the OUTPUT and INPUT attributes are used, a new file will be created which can be written to and read from. The following examples illustrate the use of the INPUT attribute:

```
OPEN #1: "#4:FILE.TEXT", INPUT
OPEN #1: "#4:FILE.TEXT", INPUT, OUTPUT
```

The UPDATE access mode indicates that the file may be read from and written to. This is the default mode. If this mode is used with the OUTPUT mode, a new file will be created which can be read from and written to (this is equivalent to using the combination of INPUT and OUTPUT access modes). The following is an example of the use of the UPDATE mode:

```
OPEN #1: "#4:FILE.TEXT", UPDATE
```

The APPEND access mode is used only with sequential files and indicates that records may be written to the end of the file. No reads may be done, nor may a RESTORE statement be used on the file. All records will be written, sequentially, starting at the end of the file. The following is an example of the use of the APPEND access mode:

```
OPEN #1: "#4:FILE.TEXT", APPEND
```

7.1.3 File Organization

Files may be opened with either of two file organization attributes: SEQUENTIAL or RELATIVE. If no attribute is specified, SEQUENTIAL is assumed. Virtual Arrays, which allow a file to be accessed as though it were an array in memory, are RELATIVE files.

SEQUENTIAL files are identical to Pascal text files. SEQUENTIAL files are written to and read from in sequential order, beginning with the first record in the file. Also, records may be appended to the end of existing SEQUENTIAL files.

Peripheral devices are treated as SEQUENTIAL files. A device may be opened and, if the device allows, written to or read from. The following examples illustrate opening the Printer (LP01 and PRINTER: are treated identically):

```
OPEN #1:"PRINTER:"
OPEN #2:"LP01"
```

An ordinary SEQUENTIAL file may be opened as follows:

```
OPEN #1:"#5:FILE.TEXT", INPUT, SEQUENTIAL
```

RELATIVE files allow sequential access and random access to the records within a file. If a RELATIVE file is to be opened, the keyword RELATIVE must be one of the specified attributes. If a new RELATIVE file is to be created, the attributes

INTERNAL and FIXED must also be specified. The following examples illustrate how relative files are opened:

```
OPEN #1:"#5:FILE1", RELATIVE, INPUT
OPEN #2:"#5:FILE2", RELATIVE, INTERNAL, FIXED, OUTPUT
```

If a record in a RELATIVE file is to be accessed out of sequence, that record is specified by a number which represents its position within the file. The first record within a file is record number zero. If it is desired to access the 12'th record in the file, the number eleven should be specified. This record number is specified within the REC clause of an INPUT, ACCEPT, RESTORE, or PRINT statement (see Section 7.2.2).

7.1.4 File Length

When a RELATIVE file is created, it is assigned a maximum file length. It may not expand beyond that size. This file length may be specified as one of the attributes. If it is not, a default size of 144 logical records will be assumed. After a file reaches its maximum size, it must be copied into a larger file before more records may beadded to it. The following is an example of file length specification:

```
OPEN #1:"#5:FILE2", RELATIVE 700, INTERNAL, FIXED, OUTPUT
```

7.1.5 File Format

The information within files may be stored in either of two formats: DISPLAY or INTERNAL.

The DISPLAY format is used with sequential files and indicates that the data is stored in ASCII format. This type of file typically contains text and string data. DISPLAY is the default file format attribute, and may be used only with SEQUENTIAL files. The following example shows the use of the DISPLAY attribute:

```
OPEN #1:"#5:F1.TEXT", SEQUENTIAL, DISPLAY, UPDATE
```

The INTERNAL format must be specified when opening or creating RELATIVE files. The INTERNAL format indicates that the data within the file is stored in binary format. This type of format is especially useful when dealing with numeric quantities. When a value is written to this type of file, it is stored in the format which corresponds to its declaration within the program, e.g. INTEGER, REAL or DECIMAL. It is important, therefore, that the data be read and written using variables of the same type. The following example illustrates the use of the INTERNAL file format attribute:

```
OPEN #2:"#5:FILE2", RELATIVE 700, INTERNAL, FIXED, OUTPUT
```

7.1.6 Record Length

Records in a SEQUENTIAL file may have a fixed or variable length. The length attribute should be specified in the OPEN statement when a file is created. This "logical record length" should be greater than or equal to 2, and less than 32767 (bytes), and must be an even number.

The VARIABLE attribute indicates that records in the file may be of different lengths. An optional maximum length may be specified by a number following the keyword VARIABLE. If no maximum length is specified, a default of 80 bytes will be assumed. If a record being written to the file exceeds the maximum record size, the current record slot is terminated, and the remaining data is written into the next record. The VARIABLE attribute is assumed if no record length attribute is indicated. The VARIABLE attibute may be used in conjunction with sequential files only. In the following example, records have a variable length with a maximum size of 200 bytes:

```
OPEN #1:"#5:F1.TEXT", SEQUENTIAL 500, VARIABLE 200, OUTPUT
```

The FIXED attribute indicates that the records in the file are all of the same size. The size is specified to the right of the keyword FIXED. Any valid INTEGER expression may be used to specify this length. An attempt to read or write records which are not of the correct size will result in an error. RELATIVE files must be created with the FIXED attribute. Also, when an existing RELATIVE file is opened, the FIXED attribute must be specified and the record size must match the size specified when the file was created. The default length for fixed records is 256 bytes. The following example creates a RELATIVE file with 32-byte records:

```
OPEN #2:"#5:FILE2", RELATIVE 700, INTERNAL, FIXED 32, OUTPUT
```

7.1.7 The ASSIGN Statement and Virtual Arrays

The ASSIGN statement is used to associate a Virtual Array with a disk file. After the ASSIGN statement has been executed, assignments may be made to the Virtual Array, in which case the disk file is written to. Also, variables may be assigned from the Virtual Array, in which case the disk file is read in order to obtain the needed values. In this way Virtual Arrays may be used as ordinary arrays, but instead of taking up space in main memory, they actually exist on disk. Very large arrays may be used in this manner, without the danger of running out of memory space. The total number of elements within an array, however, may not exceed 32767. The following is an example of the use of the ASSIGN statement to create a Virtual Array:

```
ASSIGN "#4:REAL.FILE" USING V_ARRAY(100,100)
```

In this example, REAL.FILE on disk #4: is opened and associated with V_ARRAY. V_ARRAY may now be used as any other array, but whenever it is accessed, #4:REAL.FILE is really used. V_ARRAY is declared as it would be within a DIM statement and may be preceded by INTEGER, REAL or DECIMAL. The following example shows the use of Virtual Arrays:

```
ASSIGN "#4:REALFILE" USING INTEGER V_ARRAY(100,100)
ASSIGN "#5:REALFILE2" USING STRINGS$(10,10,100)*10
V_ARRAY(99,97) = 1234
STRINGS$(2,8,97) = STR$( V_ARRAY(99,97) )
      .
      .
CLOSE V_ARRAY
CLOSE STRINGS$
```

The CLOSE statements are used to close the disk files, see Section 7.1.8.

7.1.8 The CLOSE Statement

The CLOSE statement ends the association between an opened file and its file number. The CLOSE statement is also used to end the association between a disk file and a virtual array name. The file number or virtual array name is then available to be re-used if desired. The closed disk file is inaccessible to the program unless it is re-opened. If the file number is not associated with an open file when the CLOSE statement is executed, an error will result. The EOF function can be used to determine if this association exists. The CLOSE statement has the following formats:

```
CLOSE #file_number
CLOSE virtual_array_name
```

Virtual arrays are also implicitly closed when a STOP, END, or RUN statement is executed.

When a new file is opened, it must be explicitly closed with the CLOSE statement if it is to remain on disk after the program has finished execution. By adding the word DELETE to a CLOSE statement, the closed file will be removed from the disk directory even if it existed before the program opened it. The following shows the use of the DELETE option:

```
OPEN #2:"#4:JUNK.TEXT"
CLOSE #2:DELETE
```

7.2 File I/O Statements

Records within a file may be accessed using the INPUT, ACCEPT, and PRINT statements. After the file has been opened, the execution of one of these statements causes a record to be read from or written to the indicated file. If a data separator (comma, semicolon, or apostrophe) is used to terminate one of these statements, I/O will be defered. The RESTORE statement is used to select, in a random access manner, the next record on which I/O statements will perform their function. The following are the simplest formats of these statements:

```
PRINT #file_number: variable_list
INPUT #file_number: variable_list
ACCEPT #file_number: variable
RESTORE #file_number
```

The PRINT statement outputs the variables listed to the indicated file. The INPUT and ACCEPT statements input from the file indicated, to the variables in the list. After one of these statements is executed, an internal record pointer is advanced to the next record. The RESTORE statement points the internal record pointer to the first record in the file.

If a record contains more variables than can be listed on one line, the I/O statement may be terminated with a data separator and further statements can be used to complete the I/O.

7.2.1 Sequential File I/O

Because sequential files use the DISPLAY format, variables written to them will be in the same format as if they were written to the console. In the following example the variables I, J, and K are written with trailing blanks, and a preceding blank or minus sign (because of the comma data separator, see Section 3.1). The file is then restored, and the variables are read back into a string. The Standard Functions could then be used to convert the string into three separate numeric values again.

```
PRINT #1: I,J,K
RESTORE #1
INPUT #1:THREE_VARS$
```

In the next example I, J, and K are written with commas between them (because of the apostrophe data separator, see Section 3.1). This allows the INPUT statement to read the three separate variables directly as numeric values. Alternatively, the ACCEPT statement (which doesn't treat commas as data separators) can be used to read the variables into a string.

```
PRINT #2: I'J'K
RESTORE #2
INPUT #2: I,J,K
RESTORE #2
ACCEPT #2:THREE_VARS$
```

If there are fewer variables in a record than in the variable list of an INPUT statement, subsequent records will be read until enough variables are obtained. If there are more variables in a record than in the variable list of an INPUT statement, the remaining variables will be discarded unless the variable list is terminated with a data separator. In this case the next input operation will read further data from the same record.

The following example shows how 40 variables may be written to the same record:

```
FOR I=1 TO 40 DO
    PRINT #1:DATA_ARRAY(I)'
NEXT
PRINT #1:
```

In this example, the apostrophe indicates that more data is still to be written to the same record after the current PRINT statement is finished. (The apostrophe also inserts commas between each value written to the file.) The PRINT statement after the loop completes the record. Future PRINT statements to this file will write to the next record.

The USING option (see Section 3.2.5) can be used in conjunction with the PRINT statement to files. This allows formatting of the file contents. The following is an example of the PRINT-USING statement to a file:

```
PRINT #1 USING ###.##: NUM
```

7.2.2 Relative File I/O

Relative files always contain data in the INTERNAL format. This format may be INTEGER, REAL, DECIMAL, or string. Because of this, the apostrophe data separator, which produces commas in the output, may not be used with relative files. The input variables used to read in data should be identical in number and type to the output variables used to write the data originally. Otherwise, incorrect results will occur. If string variables are written to relative files, the runtime length of the string determines the number of characters written.

A particular record in a relative file may be accessed using the REC clause. The keyword REC is followed by the number of the record (zero is the first record in the file).

In the following example two strings are written and then read from the 10'th record in a file:

```
S1$="STRING"
S2$="FELLOW"
PRINT #1, REC 9: S1$,S2$
INPUT #1, REC 9: S3$,S4$
```

Fourteen bytes are written, a length byte and six characters for both S1$ and S2$. The INPUT statement assumes the first character is a length byte and assigns S3$ accordingly. The byte which follows the first string is then assumed to be the length byte for the second string, which is assigned to S4$.

If several statements are needed to write to or read from a single record, a data separator can be placed at the end of each statement. This defers the final I/O to that record. Only the first such statement should contain a REC clause, however, since each occurrence of a REC clause will cause a new record to be accessed. The following example writes 40 values to record N:

```
PRINT #1, REC N: INFO(1);
FOR I=2 TO 40
    PRINT #1: INFO(1);
NEXT
PRINT #1
```

7.2.3 The RESTORE Statement

The RESTORE statement is used to reposition the internal file record pointer to a specific record. SEQUENTIAL files use the RESTORE statement with the following format:

```
RESTORE file_number
```

The indicated SEQUENTIAL file is repositioned to the first record within it.

RELATIVE files use the RESTORE statement with the following formats:

```
RESTORE file_number
RESTORE file_number, REC record_number
```

The indicated file's internal record pointer is set to the record number specified in the REC clause. If no REC clause is specified, the file is repositioned to the beginning. In the following example, the file is reset to the tenth record:

```
RESTORE #1, REC 9
```

The next read operation reads that record.

APPENDIX A

BASIC RESERVED WORDS

ABS
ACCEPT
ALL
APPEND
ASC
ASSIGN
AT
ATN
BASE
BELL
BREAK
CALL
CHR$
CLOSE
COS
DAT$
DATA
DECIMAL
DEF
DIM
DISPLAY
ELSE
END
EOF
ERASE
ERR
ERROR
EXP
FIXED
FNEND
FOR
FREESPACE
FTYPE
GO
GOSUB
GOTO
IF
IMAGE
INKEY
INKEY$
INPUT
INT
INTEGER
INTERNAL
LEN
LET
LIBRARY
LOG
NOT
ON
OPEN
OPTION
OUTPUT
POS
PRINT
PUNCTUATION
RANDOMIZE
READ
REAL
REC
RELATIVE
REM
RESTORE
RETURN
RND
RPT$
SEG$
SEQUENTIAL
SGN
SIN
SIZE
SPAN
SQR
STEP
STOP
STR$
SUB
SUBEND
SUBEXIT
TAB
TAN
THEN
TIME$
TO
UNIT
USES
USING
VAL
VARIABLE

APPENDIX B

BASIC ERROR NUMBERS

Compiler Errors

1. Illegal or missing label
2. Illegal or missing variable name
3. Duplicate ALL statements
4. Doubly-defined variable
5. Right parenthesis or comma is expected here
6. Bad format
7. Integer is expected here
8. No scale in decimal statement
9. Illegal character in text
10. Illegal statement
11. FOR without matching NEXT
12. Array is too large
13. Illegal variable type
14. Illegal operator in statement
15. Wrong number of dimensions
16. Wrong number of arguments to function or subroutine
17. Missing BASE in OPTION statement
18. Bad number in OPTION statement
19. Too many OPTION statements
20. Function reference is not allowed here
21. Expression should start with a constant or variable
22. Doubly-defined label
23. Too much stuff in statement
24. Missing equal-sign in assignment statement
25. Missing THEN in IF statement
26. Missing GOTO in ON statement
27. Missing equal-sign in FOR statement
28. NEXT statement without FOR statement
29. Missing TO in FOR statement
30. Undefined label
31. Colon is expected here
32. Missing ALL after ERASE
33. Left parenthesis expected here
34. REC clause is expected here
35. Too many input variables in the ACCEPT statement
36. Variable is expected here
37. String is expected here
38. Array dimension is too small
39. Pound (#) is expected here
40. Delete is expected here
41. Comma is expected here
42. File types conflict or are inconsistent
43. Modes conflict

44. USING is expected here
45. Missing exponent in number
46. FNEND not expected
47. SUBEND not expected
48. Function name not expected
49. Too many jumps
50. Too much object code
51. Out of memory space
52. Number is too large
53. Number is expected here
54. Missing GOTO or GOSUB
55. Right parenthesis or semicolon is expected here
56. Semicolon is expected here
57. Too many units are included
58. Unit not found in library
59. Error attempting to open include or uses file
60. Variable in NEXT statement does not match FOR statement
61. Too many UNIT statements
62. Too many subroutines and functions
63. Subroutine call is expected here
64. SUBEND or FNEND is expected here
65. END expected

Execution Errors

1. String size error
2. Missing or bad number
3. File is not open
4. Not enough input
5. Bad number (conversion from string)
6. Too much input
7. Too many variables for print image
8. Image field error
9. End of data list
10. Wrong type of data in data list
11. File types do not match
12. You tried to open an open file
13. You cannot restore a sequential file
14. Read record overflow of relative file
15. Write record overflow of relative file
16. Bad arguments to SEG$ function
17. Number too large for exponentiation
18. Negative argument in exponentiation
19. ON statement index is out of bounds
20. You cannot write to a read-only file
21. You cannot read from a write-only file
22. You cannot close file #0.
23. You cannot close a closed file
24. You cannot open-for-output an existing file
25. You cannot open-for-output a device
26. Relative record number is too large or too small
27. You cannot restore an APPEND file
28. The number of records in the OPEN statement is bad
29. The record size in the OPEN statement is bad
30. Too many returns from GOSUB
31. Too many GOSUB statements
32. FREESPACE argument is not zero