Cromemco

68000 Structured BASIC

Reference Manual

Cromemco

68000 Structured BASIC

Reference Manual

October 1984

023-4083
Rev. A

CROMEMCO, Inc.
P.O. Box 7400
280 Bernardo Avenue
Mountain View, CA 94039

This manual was produced using a Cromemco System Three computer running under the Cromemco Cromix Operating System. The text was edited with the Cromemco Cromix Screen Editor. The edited text was proofread by the Cromemco SpellMaster Program and formatted by the Cromemco Word Processing System Formatter II. Camera-ready copy was printed on a Cromemco 3355B printer.

TABLE OF CONTENTS

LIST OF APPENDICES

Chapter 1

INTRODUCTION

Basic is a friendly language that many programmers chose for developing small and medium-sized programs. The friendliness comes from the few restrictions placed on the programmer. For instance, when you stop program execution with the ESCAPE key, you can display and change the values of the program's variables, and then continue execution.

Cromemco 68000 Structured Basic incorporates the essential features of the Basic language and goes beyond them. During the past decade there has been an increasing awareness of the difficulties in developing complex computer programs. Once a program exceeds 1000 lines of program code the productivity of the programmer writing it falls dramatically.

The solution is to write modular programs, which increase productivity because no portion of the program is more than a few hundred lines long. A number of structured languages exist which facilitate modular programming practices. A structured language allows the programmer to create blocks of code which comprise the program. Structured languages also have program control statements that allow and insure that these blocks of code can operate independently of other blocks of code. Thus standard modules can be written that consist of many of these blocks.

Structured Basic is designed to allow these important programming practices to be used in the very popular Basic programming language. In addition, it is designed to provide the programmer with the most computing power per line of code ever offered in a version of Basic. This increase in programming power is provided to enable the programmer to quickly write programs that do more than ever before. It also assists the programmer in writing programs that are able to work correctly on the first try.

It is in programs of less than 10,000 lines of code that Cromemco Structured Basic differs from other structured languages, such as Algol and Pascal. These other languages restrict the programming practices to those appropriate for major software undertakings that might be expected to take several years of programming and involve many programmers working together and creating tens of thousands of lines of program code. The interactive power of Basic and the philosophy of its versatility and error tolerance are intentionally not implemented in these languages to discourage poor practices for large programs.

Structured Basic extends the capability of most Basics. These extensions are provided to enable programs to be written more quickly and make it more likely that these programs will work on the first try. These extensions include the following features:

1. Variable names up to 31 characters, such as "Present'worth" and "Interest'rate" may be used. All characters in the name are verified to insure uniqueness.

2. Statement labels may be referenced in addition to line numbers. **Goto Io'driver** is a valid instruction.

3. REPEAT, WHILE, IF-THEN-ELSE, and PROCEDURE statements of structured languages are provided.

4. Module handling instructions may be used to create programs that are swapped into memory from a Procedure Library file on disk. This automatic loading of software modules, each with its own local variables, makes it easy to write programs of indefinite length without regard for the computer Random Access Memory capacity.

5. Extended string functions allow the insertion of strings within strings and unambiguous conversion of strings to numeric representations.

6. A Keyed Sequential Access Method (KSAM) is provided to simplify and speed up contents-oriented file accesses such as are found in data base programming.

7. A line-oriented text editor is provided to facilitate programming changes.

8. The LVAR command is provided to enable all the variables used in the program to be listed.

9. High execution speed resulting from a semicompiling design. Execution rate is as fast for large programs as it is for small programs. This feature is not available in most other versions of Basic.

10. High accuracy 14 digit arithmetic. The binary coded decimal arithmetic used in 16K Extended Basic is essential for accounting and business programming applications.

11. Extensive string handling and output formatting capabilities. These Cobol-like capabilities make report and forms generation easy.

12. Versatile random and sequential disk access methods.

13. Automatically indented listings denoting loops and other control structures.

GETTING STARTED

Before the Structured Basic program can be used, the Structured Basic interpreter must reside in the current directory or the /bin directory.

To load the Basic program, give the following command:

sbasic68

A few seconds after you call the Basic program, the Basic sign on message will be displayed followed by the Basic prompt (>>). Basic has now been loaded into the computer's memory and is waiting for instructions.

All lines entered in Basic must be terminated by depressing the RETURN key. Basic will not respond to an instruction, accept any input, or complete a command unless it is terminated by a RETURN. If Basic does not seem to be responding as you think it should, make sure that you have properly terminated the current line (with a RETURN).

The Command or Immediate Mode

Whenever the Basic prompt (>>) is displayed, Basic is in the immediate mode. This means that an instruction (command) can be entered and Basic will respond immediately after the RETURN key is depressed.

Refer to chapter 7, "Programming Examples" for examples of simple programming functions you can perform.

Chapter 2

INSTRUCTION SYNTAX

The Cromemco Structured Basic language is designed to allow the user to structure and format a program in a wide variety of styles. This section covers the features which allow this flexibility as well as those elements of Basic syntax which do affect program operation.

SPACES OR BLANK CHARACTERS

In most cases, a space is now required after a Basic key word. The following two instructions, for example, are not equivalent.

```
GOTO10
GOTO 10
```

Basic would consider the first, GOTO10, to be a variable name. Basic would consider the second, GOTO 10, to be a legal instruction.

UPPER CASE CHARACTERS

Basic instructions can be entered in any combination of both upper and lower case characters. Upon LISTing a program, Basic will convert key words and variable names into its own format, initial character upper case, the rest lower case. REMarks and string literals will remain exactly as entered.

THE BASIC PROMPT (>>)

When Basic is ready to accept a command or statement line, it displays a prompt which consists of two greater than symbols (>>). The purpose of the prompt is to indicate that Basic has finished its last task and is waiting for the user's next instruction.

COMMANDS

A command is a Basic instruction which is executed immediately (as soon as the carriage RETURN key is pressed). Commands have no line numbers because they are not stored by Basic.

Cromemco Structured Basic allows most instructions to be used as commands. For example, Basic can be used as a calculator while it is in the command mode:

```
>>Print 20000/5
4000

>>Print (4000+77)/63.
64.714285714285
```

As can be seen, a command may be given whenever the Basic prompt (>>) is displayed.

In this manual, the term instruction includes both commands and statements. All instructions listed as instructions or commands may be used as commands, while those listed as statements may not be used as commands.

STATEMENTS

A statement is a Basic program line which contains one or more instructions, and which is stored for execution at a later time. Statements are not executed until the RUN command, or some other command which will begin execution of a program, is given. A statement line has a unique line number (within a program or Partition) by which it can be accessed. If a second statement is entered with the same line number as a line which already exists, the original line will be replaced by the new line.

Cromemco Structured Basic allows most instructions to be used as statements. For example, one Basic program can LOAD and RUN another Basic program:

```
100 Run "Prog2"
```

Execution of this statement will cause Prog2 to be LOADed into the User Area and execution to begin.

In this manual, the term instruction includes both statements and commands. All instructions listed as instructions or statements may be used as statements, while those listed as commands may not be used as statements.

Line Names

A statement line may be referenced either by a line number or by an alphanumeric line name. A line name includes from 1 to 31 characters. The first character must be alphabetic. Each of the remaining characters may be alphabetic, numeric, or the apostrophe (').

Label a line with a name by following the line number with an asterisk and line name. The asterisk and name is considered an instruction which declares the name of the line. The line may contain additional instructions separated by colons as specified by Basic syntax.

Label names are used with GOTO and GOSUB to transfer CONTROL from one part of a program to another. They may also be used to reference various parts of a program for EDITing, statement RENUMBERing, etc.

Example:

```
10 *Beginning
 .
 .
50 Gosub Get'data
60 Gosub Process'19
 .
 .
90 Goto Beginning
 .
 .
300 *Get'data : Data'pointer=1
 .
 .
360 Return
400 *Process'19 : Error'flag=0
 .
 .
450 Return
```

Multiple Instruction Lines

Cromemco Structured Basic allows more than one instruction to be associated with a single line number. Each pair of adjacent instructions must be separated by a colon (:). The number of instructions which may appear on a single line is limited only by the length of a line. For example:

```
10 *Start : Input Quan : If Quan<0 Then Goto Start
20    Print Sqr(Quan) : Print : Print
30    Rem Line 20 will print, then skip 2 lines
```

User defined functions (DEF FNs) and DATA statements must appear as a single instruction on a line. The following instructions may appear as part of a multi-instruction line but they must be the last instruction on the line, i.e., no other instruction may follow on the same line.

DELETE
ENDPROC
ENTER
ERRPROC
EXITPROC
LIST
ON-GOSUB
ON-GOTO
REM
RUN
SCR

A colon may not terminate a multiple instruction line. A colon must be followed by another instruction.

The IF-THEN instruction is unique when followed by other instructions on the same line. The reader is referred to the IF-THEN instruction for a further discussion.

Chapter 3

NUMERIC AND STRING INTERNAL MACHINE REPRESENTATION

Numeric and string alphanumeric information is stored by Basic in different forms. This section explains these different formats.

INTEGER

Integers are whole numbers and in Structured Basic they must be within the range +32767 to -32768. When stored by Basic, an integer occupies 2 bytes of memory.

Integer numbers are stored low byte first, high byte second. If the high bit of the high byte is a 1, then the number is negative; if it is a 0, then the number is positive.

A positive number is stored as the binary representation of the number.

A negative number is stored as the 2's complement of the number.

Example:

The number 1234 will be represented by the hexadecimal bytes D2 04 when stored as an Integer. When the order of the bytes is reversed (04 D2) this is the binary equivalent of the decimal number 1234.

The number -1234 will be represented by the hexadecimal bytes 2E FB when stored as an Integer. When the order of the bytes is reversed (FB 2E) this is the 2's complement of the binary equivalent of the decimal number 1234. Because the high bit of the high byte is 1, the number is negative (-1234).

SHORT FLOATING POINT

A Short Floating Point number stored by Basic occupies 4 bytes, has an accuracy of 6 digits, and must be within the range $\pm$9.99E+62 to $\pm$9.99E-65.

The first byte of a Short Floating Point number contains the sign of the number and the exponent in excess-40h (or 64 decimal) notation. If the high bit of the first byte is a 1, then the number is negative; if it is a 0, then the number is positive. Note that this is not the sign of the exponent but rather of the number itself. The remaining 7 bits of the first byte contain the exponent plus 40h. In order to find the true exponent, 40h (64 decimal) must be subtracted from this number.

The remaining three bytes contain the BCD (Binary Coded Decimal) mantissa which has been normalized to a value between 0 and 1. The implicit decimal point is located before the first byte of the mantissa. Each byte of the mantissa can contain 2 significant digits yielding a total of 6 significant digits for a Short Floating Point number.

Example:

The number -1.2345E+21 will be represented by the hexadecimal bytes D6 12 34 50 when stored as a Short Floating Point number.

The first bit of the first byte is a 1 indicating that the number is negative. The remaining 7 bits of the first byte (56h or 86 decimal) is the exponent plus 40h. To get the true exponent 40h must be subtracted from 56h. This leaves an exponent of 16h or 22 decimal. This is not the exponent of the original number because the number was normalized. In this example, normalization involved dividing the number by 10 and adding 1 to the exponent to compensate for the division.

The remaining three bytes are the BCD representation of the normalized number. BCD stands for Binary Coded Decimal which is a method of representing a decimal number in binary. Using this method, each byte can contain two one digit decimal numbers. As can be seen from the example, non-significant digits are zero filled.

LONG FLOATING POINT

A Long Floating Point number stored by Basic occupies eight bytes, has an accuracy of 14 digits, and must be within the range ±9.99E+62 to ±9.99E-65.

The internal representation of a Long Floating Point number is similar to that of a Short Floating Point number. The difference is that four additional bytes are added to the mantissa for a total of seven bytes, yielding 14 significant digits.

HEXADECIMAL

A hexadecimal number occupies two bytes and must be within the range 0h to FFFFh.

The internal representation of a hexadecimal number is the same as that of an Integer. Except for the binary functions, hexadecimal numbers are treated as signed integers. (Refer to the section "Integer" at the beginning of the chapter.)

STRING

A string is an ordered list of alphanumeric information. Examples of strings include words, sentences, parts of words, groups of letters or special characters.

Elements of a string (characters) stored by Basic occupy 1 byte each. Each character is represented internally as an eight bit number which is normally considered to be the ASCII (American Standard Code for Information Interchange) code for the character being stored. Bit number 7 (the high bit) is a parity bit in the ASCII convention (it is normally zero in Structured Basic) and, although it affects string comparisons, it does not affect the character which is PRINTed. Only GET and CHR$ can return a string byte with the high bit set (=1).

Chapter 4

CONSTANT AND STRING LITERAL FORMATS

Constants are, as the name implies, unchanging. They each have one value.

String literals are similar to constants in that they each maintain one value which does not change.

This section covers the standard formats for constants and string literals in Structured Basic.

INTEGER AND FLOATING POINT CONSTANTS

A constant is a number. It does not change value and is represented as it would be in any arithmetic computation.

There are three types of constants: integer, floating point, and hexadecimal. All constants (not specified as hexadecimal) equal to or greater than 10,000 and those containing a decimal point are always stored by Basic as floating point numbers (either Short or Long Floating Point depending on the current mode, but always Long if there are more than 6 significant digits). All constants (not specified as hexadecimal) with a value less than 10,000 and not containing a decimal point are stored by Basic as Integer numbers.

Constants

Floating Point	Integer
20000	55
3.	9985
.000376	5
12.7	458

Floating point constants (1.2, 3., 1E6, etc.) are stored according to the mode which is active when they are entered. An exception is a constant with more than 6 significant digits, which is always stored as a Long Floating Point number. For example, if the active mode is set to Short Floating Point (refer to the SFMODE instruction) and the following commands are given:

```
>>Long Long'num
>>Long'num = 1./3.
```

Long'num will be assigned a value of 0.33333300000000 because 1. and 3. are SHORT (SFMODE was current when they were entered). But, under the same circumstances, if we had said:

```
>>Long Long'num
>>Long'num = 1./3.00000000
```

Long'num would have been assigned a value of 0.33333333333333 because 3.00000000 is forced to LONG (more than 6 significant digits).

If in the two previous examples, the current mode had been Long Floating Point, Long'num would have received a value of 0.33333333333333 in both cases because both 1. and 3. would have been stored as Long Floating Point numbers.

HEXADECIMAL CONSTANTS

Hexadecimal numbers are used in base 16 arithmetic. The set of digits used in hexadecimal arithmetic is 0 through 9 and A through F.

In Cromemco Structured Basic, hexadecimal constants are identified by leading and trailing percent (%) signs. The trailing percent sign may be omitted when this will not create an ambiguous expression.

Constants

Hexadecimal	Equivalent Decimal
%8000%	-32768
%9000%	-28672
%FFFF%	-1
%9%	9
%A%	10
%F%	15
%10%	16
%80%	128
%FF%	255
%100%	256
%7FFF%	32767

A hexadecimal number is stored in the same format as an Integer and may be used wherever a constant is allowed.

STRING LITERALS

A string literal is a string which is enclosed between quotation marks. The quotation marks are not part of the string itself, but are used to delimit (mark the ends of) the string. The value of a string literal does not change.

A quotation mark can be represented within a string literal by the use of two quotation marks, one immediately following the other.

String Literals

```
"This is a string literal"
"Here are imbedded ""quotation"" marks"
"{special} *+- characters are OK!"
"-----------------------------"
"                    just spaces"
```

The trailing quotation mark on a string literal is not required at the end of a line being entered. In this case, the RETURN will terminate, but not be a part of, the string literal.

Chapter 5

VARIABLE REPRESENTATION

Variables are, as the name implies, able to change or vary in value. As a program is executed, variables may be assigned new values at any time. This section covers the standard formats for variables in Structured Basic.

NUMERIC VARIABLES

Numeric variables may be assigned numeric values. The range and accuracy of these variables depends on their type. Refer to the chapter on "Numeric Internal Machine Representation" for more information.

Format of Numeric Variables

A variable name includes from 1 to 31 characters. The first character must be alphabetic. Each of the remaining characters may be alphabetic, numeric, or the apostrophe (').

Variable names may be entered in lower or upper case characters, or any combination of the two. Basic will convert the variable name into its format which is an initial upper case character followed by all lower case characters.

The following are examples of legal variable names:

```
Interest'rate
A0
A123
Name1$
Price'to'earnings'ratio
Stock'number
```

Integer Variables

Specific variables may be set to INTEGER by the INTEGER instruction. The default mode for variables may be set to INTEGER if the IMODE instruction is given before the RUN instruction. Refer to the sections covering "Integer Internal Machine Representation", and the "INTEGER" and "IMODE" instructions.

Short Floating Point Variables

Specific variables may be set to Short Floating Point by the SHORT instruction. The default mode for variables may be set to Short Floating Point if the SFMODE instruction is given before the RUN instruction. Refer to the sections covering "Short Floating Point Internal Machine Representation", and the "SHORT" and "SFMODE" instructions.

Long Floating Point Variables

Specific variables may be set to Long Floating Point by the LONG instruction. Long Floating Point mode is normally the default mode. If the mode has been changed to SFMODE or IMODE, the default mode for variables may be reset to Long Floating Point if the LFMODE instruction is given before the RUN instruction. Refer to the sections covering Long Floating Point Internal Machine Representation, and the LONG and LFMODE instructions.

Matrices and Lists

A matrix is an array of numeric variables in a prescribed form. For example, the array:

```
 3  2  0

 1  4  6

-3  4  5
```

is a matrix with three rows and three columns. A matrix with m rows and n columns is written:

```
all   al2   al3   ...aln

a21   a22   a23   ...a2n
.
.
.
am1   am2   am3   ...amn
```

The individual entries in the matrix are called elements or cells. For example the quantity aij in the above matrix is the element in row i and column j. Subscripts used to indicate elements always denote the row first and the column second. Cromemco Basic permits the user to define one, two, or three dimensional matrices. A two (i.e., Mij) or three (i.e., Mijk) dimensional matrix is commonly called a table. A one dimensional matrix, a matrix with n columns but only one row, is commonly called a list. For example, the matrix:

```
3, -1, 5, -8
```

is a list (or a matrix) with one row and four columns.

A matrix may be defined to be composed of Long or Short Floating Point or Integer variables.

A matrix is named in the same manner as a numeric variable. A specific element of a matrix is accessed by the matrix name followed by 1, 2, or 3 indices enclosed in parentheses. These indices may be numeric expressions, variables, or constants.

Singly DIMensioned matricies are implicitly DIMensioned as 10. Larger singly DIMensioned as well as 2 and 3 DIMensional matrices must be explicitly DIMensioned by the DIMension, INTEGER, LONG, or SHORT instruction.

An element of a matrix may be used anywhere a numeric variable is allowed.

STRING VARIABLES

String variables may be assigned alphanumeric values. This includes all letters (both upper and lower case), numbers, and all printable and non-printable characters. Refer to the section on String Internal Machine Representation for more information.

Format

A string variable name includes from 1 to 31 characters. The first character must be alphabetic. Each of the remaining characters may be alphabetic, numeric, or the apostrophe ('). The last character must be a dollar sign ($).

String variable names may be entered in lower or upper case characters, or any combination of the two. Basic will convert the string variable name into its format which is an initial upper case character followed by all lower case characters.

The following are examples of legal string variable names:

```
Text$
Page$(Character)
Last'name$
Address$
```

Dimensioning String Variables

There is, for most purposes, no limit to the size (i.e., number of characters) of a string literal that may be assigned to a string variable. However, the default value in Structured Basic for string size is 11 or fewer characters. If string values of more than 11 characters are to be assigned to a variable, the string variable must be DIMensioned. The DIM statement is used in Basic to define the size of a string variable (refer to the "DIM" instruction).

Example:

```
10 Dim Apples$(20), Pears$(30), Bananas$(40)
```

In this example, the string variable Apples$ is DIMensioned to allow for strings up to 21 characters in length, the variable Pears$ is dimensioned to allow up to 31 characters, and Bananas$ is dimensioned for up to 41 characters. Any string value assigned to a variable which exceeds the specified dimension is truncated. Consequently, the programmer should be sure to DIMension string variables to handle the largest string which will be input.

Note: DIM Apples$(20) allows a 21 character string because string bytes are numbered from 0 through the specified DIM size. Remember that using the 0'th element of strings (and arrays) can save memory space.

A string of LENgth zero is called a null string.

Referencing String Variables

In many cases, the data needed from a string is contained in a subset of the string. Say, for example, a programmer wants to allow a user to enter either **Yes** or **Y** in response to a question. The programmer could do it this way:

```
      .
      .
      .
 70   Print"More data?";
 80   Input Response$
 90   If Response$="YES" or Response$="Yes" Then Goto 140
100   If Response$="yes" or Response$="Y" Then Goto 140
110   If Response$="y" Then Goto 140
120   .
      .
      .
```

By referencing just a substring of Response$, the first character in this case, the programmer could shorten this to:

```
      .
      .
      .
 70   Print"More data?";
 80   Input Response$
 90   If Response$(0,-1)="Y" or Response$(0,-1)="y" Then Goto 140
100   .
      .
      .
```

Structured Basic offers five ways to reference all or part of a string variable. While reading the description of these methods, keep these points in mind:

-- All string variables are actually one DIMension arrays. By default, the arrays are 11 characters long. Longer string variables can be declared with the DIMension instruction.

-- Each position within the array holds one character. Basic references the positions by numeric subscripts. The first position is number 0. The last position has the value of the DIMension of the string. For example, an array of DIMension 100 is referenced from position 0 through 100. Basic allows positions to be referenced with numeric constants, variables, or arithmetic expressions. These are all legal references to positions:

```
A$(6)
A$(6+2)
A$(Sum)
A$(Sum+5)
```

-- When placing data into a string variable, some of the methods of referencing positions cause Basic to first replace the data in the indicated range with null characters. Other methods do not.

The discussion of the different referencing methods use several acronyms described here:

svar

This refers to the name of a string variable.

aexp

This refers to the arithmetic expression indicating a subscripted position within the string variable.

Method 1: svar

The first method references the string variable from the first position to the last position containing a non-null character. When this method is used for assigning a value to the variable, Basic places nulls into every position and then assigns the string to the variable. The following program demonstrates this:

```
>>List

   10  Rem -- First, the variable is dimensioned
   20  Dim String$(100)
   40  Rem -- The variable is assigned one string
   50  String$="XXXXXXXXXXXXXXXXXXXXXXXXX"
   60  Print String$
   70  Rem -- When the variable is assigned a second
   80  Rem -- string, the first string is completely
   90  Rem -- erased.
  100  String$="abc"
  110  Print String$


>>Run
XXXXXXXXXXXXXXXXXXXXXXXXX
abc

***End***
```

Basic also nulls the entire array when a string variable is assigned a value with the INPUT and GET instructions.

For comparisons and output, Basic references the array from the first position to the last position containing a non-null character. Trailing nulls are ignored. This makes this method convenient for writing to a console or printer, but could cause problems when a programmer wants to write fixed-length records to a disk file.

Method 2: svar(-aexp)

The second method references the string variable from the first position to the last position in the array. This method differs from the first method in that for output, nulls trailing the last non-null character are not ignored.

In this format, aexp is any number preceded by a minus sign. Regardless of the value of aexp, the variable is referenced from the first position to the last.

When this method is used for assigning a value to the variable, Basic places nulls into every position and then assigns the string to the variable. The following program demonstrates this:

```
>>List

   10  Rem -- First, the variable is dimensioned
   20  Dim String$(100)
   40  Rem -- The variable is assigned one string
   50  String$="XXXXXXXXXXXXXXXXXXXXXXXXX"
   60  Print String$
   70  Rem -- When the variable is assigned a second
   80  Rem -- string, the first string is completely
   90  Rem -- erased.
  100  String$(-1)="abc"
  110  Print String$


>>Run
XXXXXXXXXXXXXXXXXXXXXXXXX
abc

***End***
```

Basic also nulls the entire array when a string variable is assigned a value with the INPUT and GET instructions.

For comparisons and output, Basic references the array from the first position to the last position in the array. Trailing nulls are not ignored. This makes this method useful when a programmer wants to write fixed-length records to a disk file.

A useful variation of this method initializes the entire array with a character string. The format is:

```
svar(-aexp)="string"+svar(-aexp)
```

The following example shows this format used to initialize two strings:

```
>>List

  10   Dim A$(15)
  20   Dim B$(15)
  30   A$(-1)="x"+A$(-1)
  40   Print"A$:  ";A$
  50   B$(-1)="-->"+B$(-1)
  60   Print"B$:  ";B$


>>Run
1 A$:  xxxxxxxxxxxxxxx
B$:  -->-->-->-->-->-

***End***
```

Method 3: svar(aexp)

The third method references the string variable from the position specified by aexp to the last position containing a non-null character. In this format, aexp must be a value between zero and the length of the variable. If this rule is not followed, the following will result:

-- If the subscript is less than zero, the entire string will be referenced. This is the same as using method 2 of referencing string variables.

-- If the subscript is greater than the length of the variable, a run time error message will result.

When this method is used for assigning a value to the variable, Basic places nulls into every position from aexp on and then assigns the string to the variable. The following program demonstrates this:

```
>>List
   10  Rem -- First, the variable is dimensioned
   20  Dim String$(100)
   40  Rem -- The variable is assigned one string
   50  String$="XXXXXXXXXXXXXXXXXXXXXXXXX"
   60  Print String$
   70  Rem -- When the variable is assigned a second
   80  Rem -- string, the first string is erased
   90  Rem -- from the beginning of the substring to the
   91  Rem -- end of the array.
  100  String$(10)="abc"
  110  Print String$


>>Run
XXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXabc

***End***
```

For comparisons and output, Basic references the array from the position specified by aexp to the last position containing a non-null character. Trailing nulls are ignored. This makes this method convenient for writing to a console or printer, but could cause problems when a programmer wants to write fixed-length records to a disk file.

Method 4: svar(aexp1,aexp2)

The fourth method references the string variable from one position to another. In this format, aexp1 defines the first position and must be a value between zero and the length of the variable. aexp2 defines the last position of the substring and must be a value greater than aexp1 and less than or equal to the length of the variable. If these rules aren't followed, the following will result:

-- If either subscript exceeds the length of the variable, a run time error results.

-- If aexp1 is less than zero, the entire string is referenced as though no subscript had been used.

-- If aexp2 is less than aexp1, aexp1 will be the starting character of the substring and the last character will be the last non-null character in the variable.

-- If aexp2 is less than zero, then aexp2 refers to the length of the substring instead of the position of the last character in the substring. This is the same as using method 5 of referencing string variables.

When this method is used for assigning a value to the variable, Basic does not null any portion of the substring. The following program demonstrates this:

```
>>List
   10  Rem -- First, the variable is dimensioned
   20  Dim String$(100)
   40  Rem -- The variable is assigned one string
   50  String$="XXXXXXXXXXXXXXXXXXXXXXXXX"
   60  Print String$
   70  Rem -- When the variable is assigned a second
   80  Rem -- string, none of the variable is nulled.
  100  Rem -- Note that the second string is only
  110  Rem -- 8 characters long although 11 positions
  120  Rem -- in the variable potentially could be
  130  Rem -- affected.
  150  String$(5,15)="@@@@@@@@"
  160  Print String$
  170  Rem -- Again, a string is assigned to the variable
  180  Rem -- and nothing is nulled
  190  String$(5,15)="-----"
  200  Print String$

>>Run
XXXXXXXXXXXXXXXXXXXXXXXXX
XXXXX@@@@@@@@XXXXXXXXXXXX
XXXXX-----@@@XXXXXXXXXXXX

***End***
```

For comparisons and output, Basic references the array from the position specified by aexp1 to aexp2.

Method 5: svar(aexp1,-aexp2)

The fifth method references the string variable from one position for a specified number of positions. In this format, aexp1 defines the first position and must be a value between zero and the length of the variable. aexp2 specifies the length of the substring. The minus sign must appear before the expression even

though it doesn't denote a negative number. If these rules aren't followed, the following will result:

-- If either subscript exceeds the length of the variable, a run time error results.

-- If aexp1 is less than zero, the entire string is referenced as though no subscript had been used.

-- If the sum of aexp1 and aexp2 minus one is greater than the length of the variable, a run time error results.

When this method is used for assigning a value to the variable, Basic does not null any portion of the substring. The following program demonstrates this:

```
>>List
   10  Rem -- First, the variable is dimensioned
   20  Dim String$(100)
   40  Rem -- The variable is assigned one string
   50  String$="XXXXXXXXXXXXXXXXXXXXXXXXXX"
   60  Print String$
   70  Rem -- When the variable is assigned a second
   80  Rem -- string, none of the variable is nulled.
  100  Rem -- Note that the second string is only
  110  Rem -- 8 characters long although 10 positions
  120  Rem -- in the variable potentially could be
  130  Rem -- affected.
  150  String$(5,-10)="@@@@@@@@@"
  160  Print String$
  170  Rem -- Again, a string is assigned to the variable
  180  Rem -- and nothing is nulled
  190  String$(5,-10)="-----"
  200  Print String$


>>Run
XXXXXXXXXXXXXXXXXXXXXXXXX
XXXXX@@@@@@@@@XXXXXXXXXXXX
XXXXX-----@@@XXXXXXXXXXXX

***End***
```

For comparisons and output, Basic references the array from the position specified by aexp1 to aexp2.

Examples of Referencing Substrings

Assume that the command Wd$ = "ABCDEFGHIJK" has been given and that the string variable Wd$ has not been DIMensioned in a DIM instruction. (That is, it is implicitly DIMensioned at 10 and it contains 11 characters numbered zero through ten.)

Instruction	Result	Explanation
Print Wd$	ABCDEFGHIJK	Method 1 was used to PRINT the entire string.
Print Wd$(5)	FGHIJK	Method 3 was used to PRINT the substring starting with character 5 (remember that the characters are numbered starting with 0).
Print Wd$(4,8)	EFGHI	Method 4 was used to PRINT the substring starting with character 4 and ending with character 8.
Print Wd$(3,-2)	DE	Method 5 was used to PRINT the substring starting with character 3, for a length of two characters.

Now assume the command Wd$ = "ABCDEFG" has been given. This will fill the string with the seven characters A-G and 4 null characters. In the following examples the lower case n represents a null character.

Instruction	Result	Explanation
Print Wd$	ABCDEFG	Method 1 was used to PRINT character 0 through the first non-null character.
Print Wd$(-1)	ABCDEFGnnnn	Method 2 was used to PRINT characters 0 through Dim.
Print Wd$(8)	nnn	Method 3 was used to PRINT characters 8 through 10. Note that character 8 is greater than the LENgth of Wd$ so that all succeeding nulls are referenced by implication.
Print Wd$(5)	FG	Method 3 was used to PRINT character 5 through the last non-null character.
Print Wd$(5,2)	FGnnnn	Method 4 was used with aexp2<aexp1 so the string was output from character 5 through the DIMension of the string.
Print Wd$(8,9)	nn	Method 4 was used to output characters 8 through 9.
Print Wd$(13,15)		Method 4 error, since aexp1>Dim.
Print Wd$(3,%8000%)		A length of -0 indicates a null string, no string is output.

Chapter 6

OPERATORS

An operator is a symbol or group of letters which indicates that an action is to be taken on one or two items. This section describes the types of operators used in Structured Basic and the symbols which represent them.

ARITHMETIC OPERATORS

The arithmetic operators are analogous to their algebraic counterparts:

Arithmetic Operator	Meaning
+	plus sign (positive number)
-	minus sign (negative number)
** or ^	exponentiation
*	multiplication
/	division
+	addition
-	subtraction

Arithmetic operations in a numeric expression are performed, according to the priority of the operation, from left to right. All operations enclosed within parentheses are performed first. When multiple sets of parentheses appear, the operations in the innermost set of parentheses are performed first, followed by the operations in the next set of parentheses, and so on until the operations in the outermost set of parentheses are evaluated. Following evaluation of expressions enclosed in parentheses, arithmetic operations are performed in the following order: plus and minus signs (unary operators), exponentiation, division and multiplication (these two operations have the same priority), and addition and subtraction (these operations also have the same priority). When operations have the same priority, calculations are performed from left to right within the expression. The use of parentheses can alter the order in which operations are performed since the parentheses override both the left to right priority and the normal order of operations.

Examples:

```
A + B * C ** D
```

The above expression will be evaluated as follows:

```
        temp = C ** D
       temp1 = temp * B
final value = temp1 + A
```

The order of association can be changed by the use of parentheses:

```
((A + B) * C) ** D
```

This expression will be evaluated as follows:

```
        temp = A + B
       temp1 = temp * C
final value = temp1 ** D
```

It is a good idea to use parentheses if there is any doubt as to the order in which a series of operations will be performed. Intermediate results may also be assigned to temporary variables if this will help to clarify the order of operations.

ASSIGNMENT OPERATOR

The equal sign (=) is the assignment operator. It is used to assign the value of an arithmetic, relational, or Boolean expression, or a function, to a numeric variable.

In addition, the assignment operator is used to assign the value of a string literal, string variable, or string function to a string variable.

Basic also uses the equal sign as a relational operator (see the following section). The only place the equal sign is a legal assignment operator is the first equal sign in a LET, implied LET, or MAT instruction.

Examples:

```
100  Number = 2
110  Print Number
120  Number = Number*2
130  If Number >= 16384 Then Stop
140  Goto 110
```

This program will print out all of the powers of 2 which are less than 16384 and then it will Stop. Statement 100 assigns the value of 2 to the numeric variable Number.

Line 120 assigns the result of the multiplication of the variable Number times the constant 2 to the variable Number. This is a good example of the reason the term assigned to is used instead of equal to. Number is obviously not equal to Number times 2. The expression on the right side of the assignment operator is evaluated (Number * 2) and then this value is assigned to the variable on the left of the assignment operator.

Statement 130 uses the greater than or equal to sign as a relational operator. The relational expression (Number >= 16384) is evaluated as true or false. If it is true, the portion of the instruction following Then is executed (program execution STOPs). If it is false, the next statement (140) is executed. Refer to the following section for a further discussion of relational operators.

Line 140 transfers CONTROL back to line 110 and execution continues until the relational expression (Number >= 16384) is true.

```
10   Termination'cond$ = "End"
20   *Loop : Print "Enter a name,"
30   Print "End to stop";
40   Input Name$
50   If Name$ = Termination'cond$ Then Stop
60   Print Name$ : Print
70   Goto Loop
80   End
```

Statement 10 assigns the value of the string literal "End" to the string variable Termination'cond$. Line 40 gets a string from the user and assigns the value of that string to the string variable Name$ after having set Name$ equal to a string of null characters (see the INPUT instruction).

Statement 50 uses the equal sign as a relational operator. The relational expression (Name$ = Termination'cond$ or we could have used Name$ = "End") is evaluated as true or false and the execution of the program continues on the basis of the evaluation. See the next section for more information on relational operators.

RELATIONAL OPERATORS

Relational Operator	Meaning
=	is equal to
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to
<> or #	is not equal to

Relational operators are used to compare two expressions. Each of these expressions may be composed of other relational, Boolean, or arithmetic expressions. This allows the user to nest relational and Boolean expressions.

In addition, relational operators are used to compare string variables, string literals, and string functions with each other.

The result of a relational operation is either true (=1) or false (=0).

Example:

```
100     If Test<=0 Then Goto 150
110     Print Test=0
120     Result=Test<0
130     Boolean=Test=5
150     End
```

Statement 100 can be read as, "If the value of the variable Test is less than or is equal to zero, transfer program control to line 150; otherwise continue with the next statement (line 110)." This is the most common use of a relational operator. This statement can be used to test the validity of user input, transferring CONTROL to another section of code if the input is not as desired.

Statement 110 may at first look like an improper assignment instruction. Remember that the equal sign is, in this case, a relational operator, not an assignment operator. This statement can be read as, "Compare the value of Test with 0 and PRINT the result of the comparison." Thus, a zero (=false) will be output if Test is not equal to zero and a 1 (=true) will be output if Test is equal to zero.

Statement 120 is similar to 110, with the exception that the result of the relational comparison is assigned to the variable Result. Result will therefore take on a value of 0 or 1.

Statement 130 is a combination of the ideas used in 110 and 120. Remember that the first equal sign is an assignment operator while the second equal sign is a relational operator. This statement can be read as, "Compare the value of the variable Test with five and assign the result of that comparison to the variable Boolean". Boolean will take on the value of 1 (=true) if Test is equal to five, otherwise Boolean will be set equal to 0 (=false).

Example:

```
300     Large$ = "zzz"
305     *Get'string : Input Test$
310     If Test$ > Large$ Then Goto Get'string
315     Large$ = Test$
320     Print Large$
325     Goto Get'string
```

Line 300 initializes the string variable Large$ by filling it with z's (the alphabetic character with the highest value in the ASCII collating sequence). Line 305 requests string data from the user.

Statement 310 evaluates the relational expression (Get'string$>Large$) on the basis of the ASCII collating sequence (refer to the table of ASCII characters in the appendix). If the first character in the string Get'string$ is different from the first character in the string Large$ then these characters are compared and this relationship determines the relationship of the two strings. If the first character in each string is identical to the other, the comparison moves on the second character and so on. For example:

Value of Get'string$	**Value of Large$**	**Value of Expression Get'string$>Large$**
George	Fred	false
Fred	April	true
April	Ted	false
april	Ted	true
april	ted	false
april	apron	false
april	apricot	true
april	aprilandmay	false

Notice that <u>all</u> lower case letters follow the set of upper case letters in the ASCII collating sequence and that if two strings are the same except for length, the longer string has a greater value than the shorter one.

Continuing with the example, if the relational expression (line 310) is evaluated as true, CONTROL is transferred to line 305 and the user is asked for another string. If the expression is false, this means that the string which was just INPUT by the user is closer to the beginning of the ASCII collating sequence

than any other string which had been INPUT previously. The value of this string is then assigned to the string variable Large$ so that all strings which are INPUT following this one can be compared with it. The value of the string is then PRINTed and CONTROL is transferred to line 305. Execution of this program may be terminated by depressing the ESCAPE key.

BOOLEAN OPERATORS

Boolean operators perform a logical operation on one or two expressions. The expressions in a Boolean operation may take on one of two values: false (=0) or true (=1). In Cromemco Structured Basic, all values which are not equal to zero (false) are considered to be true when used with Boolean operators.

Boolean Operator	Meaning
And	logical And
Or	logical Or
Xor	logical eXclusive Or
Not	logical Not or negation

And Boolean Operator

The And Boolean operator compares two logical values and if both are 1 returns a result of 1. If both values are not 1, then the result is 0.

Truth Table

AND	0	1
0	0	0
1	0	1

Or Boolean Operator

The Or Boolean operator compares two logical values and if either or both are equal to 1 then the result is 1. Otherwise the result is 0.

Truth Table

OR	0	1
0	0	1
1	1	1

Xor Boolean Operator

The Xor Boolean operator returns a 0 if the logical values are identical and a 1 if the logical values are not identical.

Truth Table

XOR	0	1
0	0	1
1	1	0

Not Boolean Operator

The Not Boolean operator returns the complement of any logical value. In other words, if the logical value is 1, the Not operator returns a 0. If the logical value is 0, a 1 is returned.

Truth Table

Not 0 is 1

Not 1 is 0

Examples:

```
10 True = 1 : False = 0
11 If True And True Then Print "11 True"
12 If Not True Then Print "12 True"
13 If Not False Then Print "13 True"
14 If True Xor True Then Print "14 True"
15 If Not (True Xor True) Then Print "15 True"
16 If True Or False Then Print "16 True"
```

The statements above give some examples of the use of Boolean operators. The IF-THEN instructions are used in these examples to test if a given Boolean expression is true or false. If the expression is true, the PRINT instruction following the IF-THEN is executed. If it is false, then nothing is printed.

The results of each of these expressions can be determined from the preceding truth tables. For example, line 14 uses the XOR operator to compare two true (=1) values. Looking at the XOR Truth Table, it can be seen that a 1 and a 1 yield a 0 or false value. Line 15 negates this same expression. Looking at the Not Truth Table, NOT 0 (Not false) is seen to be equal to 1 or true.

The results of the rest of the examples can be determined in a similar manner, then tested on the computer.

Chapter 7

PROGRAMMING EXAMPLES

The examples in this chapter are intended for the first time user. They explain in step by step detail the procedures for creating, editing, and saving a Basic program. If a more detailed description of a Basic instruction is required, the reader is referred to the later chapters of this manual which cover the instructions on an individual and in depth basis.

Additional programming examples for the more advanced user are included throughout the manual.

The first instruction we will discuss is PRINT. PRINT causes Basic to display the information following the word PRINT. Using only the PRINT instruction, and the standard Arithmetic Operators (+ for addition, - for subtraction, / for division, and * for multiplication), Basic may be used as a calculator.

Examples:

Instruction	**Explanation**
Print 5	This will PRINT the number 5. Since there is nothing following the 5, Basic will add a RETURN and LINE FEED after the 5 so that the next item which is PRINTed will start in column 0 of the following line.
Print 7 + 4	This will PRINT the result of adding 7 and 4, or 11.
Print "FRED"	This will PRINT the word FRED. Notice that if a group of letters (called a string) are to be PRINTed, they must be enclosed in quotation marks. When a string is enclosed in quotation marks, it is called a string literal. Numbers (called constants) which are to be used in a computation may not be enclosed in quotation marks.
Print	This will PRINT a blank line.
Print 1,2,3,4	This will PRINT the numbers 1 through 4 in four columns across the screen. When commas (,) are used to separate items in a PRINT list, the items are aligned in four columns across the console screen. If semicolons (;) are used, the items are displayed with no intervening spaces.
Print "ANS. = ";75	This will display ANS. = 75 on the console. Notice that a blank was included after the equal sign in the string literal. This blank is considered part of the string literal, just as any other character.

Next we will discuss variables and the Assignment Operator. A variable is the name of a location in the computer's memory. While the name of a variable stays the same, the contents of the variable may vary.

A variable name may be thought of as a label which has been affixed to a box. If we are speaking of an arithmetic variable, then the box would contain a number. The number inside the box is the value of the variable. The value of the variable can be changed while the name of the variable stays the same. Similarly, a string variable would contain a string of characters which could include letters, spaces, numbers, and any other printable characters. This combination of characters would be the value of the string variable.

Arithmetic variable names are composed of a letter (A-Z) followed by any combination of letters, numbers, and apostrophes not exceeding 31 characters in length. Some examples of arithmetic variable names are:

```
Name
Time'of'day
Table'of'values
Social'security'number
```

String variables may use any name which is a legal arithmetic variable name. A dollar sign, however, must immediately follow the name. Some examples of string variable names are:

```
Name$
Address$
Vegetable$
License'plate$
```

The assignment operator assigns a value to a variable. The equal sign (=) is used for this purpose. In Basic, the equal sign can be read as, "is assigned the value of." Notice that it does not necessarily denote equality.

Examples:

Instruction	**Explanation**
Name$ = "FRED"	The value of the string literal "FRED" is assigned to the string variable named Name$. Notice that, although the string literal "FRED" must be enclosed in quotation marks, the quotation marks are not part of the string variable. The quotation marks indicate to Basic that the enclosed characters are to be considered a string literal.
Print Name$	This instruction will display the value of Name$ on the console. If this instruction follows the preceding one, FRED will be PRINTed on the console terminal.
P4 = 775	The variable named P4 is assigned the value of 775.
P4 = P4 + 1	P4 is assigned the value of P4 + 1. If this instruction follows the previous example, P4 will have the value of 776.
Text$ = " IS NO. "	The string variable Text$ is assigned the value of the string literal " IS NO. ".
Replace$ = Text$	The string variable Replace$ is assigned the value of the string variable Text$.
Print Name$; Replace$; P4	This will print FRED IS NO. 776 (assuming it follows the above assignment instructions).

The next instruction we are going to discuss is the INPUT instruction. When executed, this instruction displays a question mark on the console and waits for the user to supply a number or a string.

Examples:

Instruction	**Explanation**
Input Age	This command will display a question mark and wait for the user to enter a number. The number which is entered will be assigned to the variable named Age.
Input Month$	This command will display a question mark and wait for the user to enter a string (any group of characters, in this case up to a maximum of eleven). The value of the string which is INPUT will be assigned to the string variable named Month$.

The RUN or Program Execution Mode

Up to this point, all of our examples have been executed as commands or in what is referred to as the immediate mode. Each command was executed as soon as the RETURN key was depressed. Once executed, the entire instruction had to be typed again in order to be re-executed.

Now we shall use these same instructions to write a program. A program consists of statements. A statement is nothing more than an instruction with a line number preceding it. Line numbers are also called statement numbers; the two terms are interchangeable. The line number indicates to Basic that the instruction is to be stored in memory for later execution. A line number (or line name) is also a useful way to refer to a Basic statement if it needs to be changed.

Let's take a moment to cover the rules for naming variables. A variable name must start with a letter (A through Z) which may be followed by up to 30 more characters. These subsequent characters may be letters (A through Z), numbers (0 through 9), or the apostrophe ('). The apostrophe has no different significance than the letters and numbers. It is very useful for breaking up long variable names so that they may be read more easily. Consider these examples:

Timeofday	or	Time'of'day
Firstrecordprimarykey	or	First'record'primary'key
Readarecord	or	Read'a'record
Getabandcblocks	or	Get'a'b'and'c'blocks

Try to make your variable names reflect the meaning of the contents of the variables. This will make your program easier to debug and will also make it easier for another person to understand.

Now we can enter a program:

```
>>100 Input Name'of'person$
>>200 Print Name'of'person$;" is smart!"
```

This program is now current in the User Area. The User Area is the Basic workspace in which a program can be written, EDITed, and RUN. The LIST command displays the contents of the User Area.

The RUN command causes Basic to execute the program in the User Area. Program execution begins with the statement with the lowest line number and continues sequentially.

We can now execute the program:

```
>>Run
? Ed
```
Basic displays the ? (prompt) and the user enters a string. Don't forget the RETURN at the end of the string.

```
Ed is smart!
```
Basic PRINTs the string variable (Ed) and the string literal (is smart!).

```
***End***
```
Basic tells you that it's done with the program,

```
>>
```
and prompts you that it is ready for additional instructions.

To get a LISTing of your program, type the command LIST in response to the Basic prompt.

```
>>List

  100 Input Name'of'person$
  200 Print Name'of'person$;" is smart!"

>>
```

Program Editing

Continuing with the same program, suppose we wish to alter a statement.

To change a line, it can be typed over again. When Basic recognizes that it already has a line with the same line number, it will replace the old line with the new one. To delete a line, type the line number followed by a RETURN.

We can also use the Basic In-Line Editor. The Editor facilitates both minor corrections to long lines and multiple replacements. Suppose we wanted to change line 200 of the program above:

```
>>Edit 200

-  200    Print Name'of'person$;" is smart!"
:                                    i very
-  200    Print Name'of'person$;" is very smart!"
:                                           i!!
-  200    Print Name'of'person$;" is very smart!!!"
:RETURN

>>
```

The EDIT command can be given with or without a line number. When used without a line number, each line of the program will be displayed in turn for EDITing. When a line number is given, only the specified statement line will be displayed.

The Editor precedes a line to be EDITed with a dash (-) and prompts the user with a colon (:). Following the colon the user can position the cursor under the part of the line to be EDITed by typing spaces.

In our example above, line 200 was displayed followed on the next line by the colon prompt. Spaces were typed until the cursor was just under the space before smart. Then **i** (for insert) was typed, followed by the text to be inserted. Notice that an insertion precedes the character above the **i**. The text to be inserted was followed by a RETURN. Basic displayed the line again, incorporating the change we just made. Next, some exclamation points were added to the end of the line in a similar manner. This time after Basic displayed the corrected line and displayed the Editor prompt (:) a RETURN alone was entered to indicate that no further changes were to be made.

While using the EDIT command, the character **d** may be positioned under any characters in a line which are to be deleted. As many **d**'s must be typed as there are characters to be deleted. After the RETURN key is pressed, the line will be displayed as EDITed. A RETURN alone in response to the EDIT prompt will return the user to the Basic monitor.

```
>>List
```

After making the change, we can list the program to make sure that we changed it properly.

```
  100 Input Name'of'person$
  200 Print Name'of'person$;" is very smart!!!"

>>
```

Again, Basic waits for further instructions.

Suppose that, instead of executing these statements only once, we wanted them executed several times. We could type RUN each time the program was to be executed, or we could add another statement which would direct Basic back to the beginning of the program each time it finished PRINTing:

```
>>300 Goto 100
```

This GOTO statement tells Basic to GOTO line number 100 and continue execution from there.

```
>>List

  100 Input Name'of'person$
  200 Print Name'of'person$;" is very smart!!!"
  300 Goto 100

>>Run
? Alice
```

Don't forget the RETURN.

```
  Alice is very smart!!!

? Eileen
```

After it was done with statement 200, statement 300 told Basic to go back to statement 100, which gave us another question mark.

```
  Eileen is very smart!!!

?
```

This could (and would) go on forever. This can be avoided by depressing the ESCAPE key (appropriately named). On a Cromemco terminal this key is located on the upper left of the keyboard and is marked ESCAPE.

```
***100 ESCAPE***
```

Basic tells you what happened,

```
>>
```

and waits for further instructions.

EXAMPLE PROGRAM

Let's write a program to compute a person's age in the year 1995. For this program, we will introduce two new instructions. The first of these is SCR which is short for SCRatch. This instruction clears memory of a program or any statements which may be in the User Area:

```
>>Scr

>>List          Nothing will be listed, because everything has been
                SCRatched.

>>
```

The other instruction is AUTOL which is short for AUTOmatic Line numbering. This is a convenient feature of Cromemco Structured Basic which allows the programmer to concentrate on the program and forget about entering line numbers. AUTOL is followed by two numbers (called arguments). The first of these indicates the first line number, while the second indicates the increment between line numbers. In the following example, the line numbers are automatically typed by Basic while the program is entered by the user:

```
>>Autol 1000,10

>>1000 Print"Enter the year of your birth: ";
>>1010 Input Birth'year
>>1020 Print"In 1995 you will be ";1995-Birth'year;
>>1030 Print " years old"
>>1040 Print
>>1050 Goto 1000
>>1060
>>
```

There are several things about this program which bear discussion. The first is that the AUTOmatic Line numbering mode may be terminated by entering a RETURN alone in response to a line number, as was done on line 1060. Line 1020 and 1030 PRINT two strings with a number in the middle. The number is the result of a computation, the numeric variable Birth'year is subtracted from the numeric constant 1995. Line 1040 is included so that there will be a blank line PRINTed between examples.

Listing to a Disk File

The program we have written will stay in memory (in the User Area) as long as we don't execute the SCRatch or BYE command or turn the power off.

We may write the program to a disk file so that it may be retrieved at a later time. Once it has been LISTed on the disk, we may SCRatch the User Area or turn off the system power and our program will still be safe on the disk.

The only way a disk file can be destroyed is by ERASing it or by writing another file with the same name to the disk. In the latter case, the second file would be written over the first so that the first file would be destroyed.

The LIST command, as we have been using it, LISTs the contents of the User Area to the console. If the LIST command is followed by a string literal, Basic will interpret the string literal as a file name, and LIST the contents of the User Area to that file. Assuming that our program is still in the User Area, the following command will write the program to a file called FIRST.

```
>>List "First.lis"

>>
```

To get a list of all the files on the disk enter the command:

```
>> Dir
```

Basic will respond with a list of file names, including FIRST.LIS. Now you can turn off the computer without losing your program.

Note: Remember to exit Basic and remove the disk before turning the power off. It is also a good idea to insert the disk only after the power has been turned on. Do not turn the power on or off while the disk is inserted in the machine.

Reading a Program from a Disk File

A program which has been LISTed to a disk file may be re-entered by using the ENTER command.

Before ENTERing any program, whether from a disk file, or from the console, it is a good idea to SCRatch the User Area. Assuming that we are once again (or still) in Basic, we can give the following two commands to clear the User Area and ENTER our program which was LISTed to the disk file FIRST.LIS.

```
>> Scr

>> Enter "First.lis"

>>
```

Our program may now be LISTed, RUN, or EDITed as we desire.

Using the SAVE and LOAD commands

The SAVE command will write the program which is current in the User Area to a disk file. The program will be in internal machine format unlike a program which is written to disk using the LIST command, which is in text format. The LOAD command is to the SAVE command what the ENTER command is to the LIST command. LOAD allows the user to read a SAVEd file into the User Area. Notice that LOAD cannot be used with a LISTed file and that ENTER cannot be used with a SAVEd file.

The formats of these commands are:

```
>>Save "Second.sav"
>>Load "Second.sav"
```

When used with larger programs, SAVE and LOAD can be significantly faster than LIST and ENTER. Also, when the RUN instruction is given with the name of a SAVEd file, the specified file will be LOADed and RUN with only that one instruction. Programs may be chained in this fashion, one calling the next, calling the next, etc.

```
>>Run "Second.sav"

or

9999 Run "Second.sav"
```

The execution of either of these instructions will cause the SAVEd program SECOND.SAV to be LOADed into the User Area and execution to begin.

Note in the above examples the use of the three-letter LIS or SAV extension to the filename. This extension to the name is not required, but is good practice for ease in remembering the type of file later on. It is conventional to use the file extensions "LIS" or "LST" for a LISTed file and the extension "SAV" for a SAVEd file.

STATISTICAL ANALYSIS PROGRAM

A common use for a computer is the calculation of statistics for a set of data. This program will compute six common statistics for any set of data which the user may enter.

```
100  Count=0 :  Sum=0 :  Sum'of'squares=0
110  Input "Number (9999 to stop):  ", Number
120    While Number#9999
130    Count = Count + 1
```

```
140     Sum = Sum + Number
150     Sum'of'squares = Sum'of'squares + Number**2
160     Input "Number (9999 to stop):  ", Number
170     Endwhile
180   Rem ************************
190   Rem All data in, compute statistics.
200   Mean = Sum/Count
210   Temp = Count*Sum'of'squares
220   Variance = ((Temp-(Sum*Sum))/Count)/(Count-1)
225   If Variance <0 Then Variance =0
230   Standard'deviation = Sqr(Variance)
240   Rem ************************
250   Rem All statistics computed, PRINT results
260   Rem Call a user defined function to round
270   Rem results to 4 decimal places.
280   Print "Number", Count
290   Print "Sum", Fnround(Sum)
300   Print "Sum of squares", Fnround(Sum'of'squares)
310   Print "Mean", Fnround(Mean)
320   Print "Variance", Fnround(Variance)
330   Print "Standard'deviation", Fnround(Standard'deviation)
340   Rem
350   Rem
360   Rem
370   Def Fnround(X) = Int(X * 10000.  + 0.5)/10000.0
380   End
```

Line 100 initializes three of the program variables. These three variables (Count, Sum, and Sum'of'squares) are used to hold sums of values as the program is executed. They must be set equal to zero at the beginning of the program in order to assure that the values they sum are accurate.

Lines 110 through 170 will continue to accept numbers from the user, keeping track of how many numbers have been INPUT (Count), the sum of the numbers (Sum), and the sum of the squares of the numbers (Sum'of'squares), until the number 9999 is entered. For the purpose of this example, it is assumed that 9999 is outside the range of numbers which will be INPUT into this program.

CONTROL will remain within the WHILE loop WHILE the variable Number is not equal to (#) 9999. This means that statements 130 through 170 will be executed over and over again WHILE Number is not equal to 9999. As soon as the user enters 9999, the variable Number is assigned the value 9999, the condition for the WHILE loop is false (Number now equals 9999), and CONTROL is passed to the statement following the ENDWHILE.

Now the rest of the statistics are computed and all results are PRINTed.

Notice that the user defined function Fnround is used to round the answers to 4 decimal places for PRINTing.

Chapter 8

INSTRUCTIONS FOR DEVELOPING A PROGRAM

This chapter describes instructions used for developing programs. These include instructions to:

-- Edit, change, and delete lines
-- Save programs on disk and retrieve them
-- Renumber lines in programs
-- Execute programs
-- Debug programs

command: **Automatic Line Numbering**

format: **AUTOL L1,L2 *AUTOL**

where:

L1 is the starting line number.

L2 is the line number increment.

The AUTOL command provides automatic statement line numbering so that the user does not have to enter a line number for each line when entering a program.

Note:

1. The automatic generation of line numbers may be terminated by pressing the ESCAPE or RETURN key when the user is prompted for the next line.

Example:

```
>>Autol 100,10

>>100 Rem All line numbers in this
>>110 Rem example are generated by Basic.
>>120 Index,=5
             $
Error 1 -- Syntax
>>120 Index=5
>>130 Rem Notice that after a Syntax Error
>>140 Rem Autol will re-prompt so that
>>150 Rem the error can be corrected.
>>160 Print Index
>>170 End
>>180
>>
```

instruction: **Bye**

format: **[Ln] BYE**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The BYE instruction is used to exit from Basic and return to the operating system.

Notes:

1. After the BYE instruction is executed the computer will respond with the operating system prompt.

2. BYE will close all files which are OPEN at the time the instruction is executed.

Example:

```
>>Bye

%
```

In this example, the user has typed the BYE command. The next prompt displayed (%) indicates that the user has returned to the Cromix shell.

instruction: **Delete Statement Lines**

format: **[Ln] DELETE L1**

[Ln] DELETE L1,

[Ln] DELETE L1,L2

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

L1 — is a line number or line name.

If L1 is the only argument and there is no comma following it, then L1 is the only line DELETEd.

If L1 is the only argument and the comma is included, L1 through the last line in the program are DELETEd.

L2 — is an optional line number or line name which indicates the last line to be DELETEd. If included, it must be preceded by L1 and a comma.

The DELETE instruction is used to remove statement lines from the program currently in the User Area.

Notes:

1. The DELETE instruction must be the last (or only) instruction on a line.

2. The DELETE instruction must have at least one argument (L1).

Example:

```
>>List

   10    Input Num1,Num2,Num3
   20    Sum = Num1+Num2+Num3
   30    Print Num1
   40    Print Num2
   50    Print Num3
   60    Print Sum
   70    End

>>Delete 30,50

>>List

   10    Input Num1,Num2,Num3
   20    Sum = Num1+Num2+Num3
   60    Print Sum
   70    End
```

Here the DELETE command removed lines 30 through 50 from the program.

command: **List Disk Files**

format: **DIR**

DIR svar

where:

svar is a string variable or a string literal file reference.

The DIR command corresponds to the Cromix ls command (see the Cromix Manual for a full description). The DIR command lists files either in the current directory or disk drive, giving the size of each file in Kilobytes. If the optional file pathname reference is used, it must be enclosed in quotation marks (string literal) or else must be a valid string variable.

Examples:

Dir	will list all files in the current directory.
Dir "*.SAV"	will list all files in the current directory with the file name extension SAV.
Dir "/usr/basic"	will list all files in the directory /usr/basic.

command: **Edit Program Lines**

format: **EDIT**

EDIT L1

EDIT L1,

EDIT L1,L2

where:

L1 is an optional line number or name.

If L1 is omitted, all lines of the program are processed.

If L1 is the only argument and there is no comma following it, then L1 is the only line processed.

If L1 is the only argument and the comma is included, L1 through the last line in the program are processed by the command.

L2 is an optional line number or name which indicates the last line to be processed. If included, it must be preceded by L1 and a comma.

EDIT is one of the three commands which are collectively called the Cromemco In-Line Basic Editor. The other two are CHANGE and FIND. EDIT lists lines of code one at a time, as specified by the arguments L1 and L2. Each line to be EDITed is preceded by a dash (-) and followed on the next line by a colon (:) prompt. In response to the prompt, the user may type:

1. A RETURN, which goes on to the next operation, accepting any changes which may have been made to the line, or

2. An ESCAPE, which leaves the line unchanged and exists EDIT mode, or

3. A series of spaces, which will position the cursor under the character which is to be EDITed.

   At this point, the user can give the following commands:

   a. **D** (Delete) - deletes the character above the cursor. Several deletions can be made on one line.

   b. **I** (Insert) - inserts the string which follows. An insertion can follow one or more deletions. An insertion precedes the character above the I.

c. **K** (Kill) - deletes the rest of the line from the current position of the cursor.

Upon issuing the EDIT command, the edited line is typed out and a prompt is given for changes.

When the line has been changed to the user's satisfaction, a RETURN should be typed in response to the colon(:) prompt to exit EDIT mode.

command: **Find String**

format: **FIND**

FIND L1

FIND L1,

FIND L1,L2

where:

L1 is an optional line number or name.

If L1 is omitted, all lines of the program are processed.

If L1 is the only argument and there is no comma following it, then L1 is the only line processed.

If L1 is the only argument and the comma is included, L1 through the last line in the program are processed by the command.

L2 is an optional line number or name which indicates the last line to be processed. If included, it must be preceded by L1 and a comma.

FIND is another of the interactive Editor commands. FIND will locate all occurrences of a string within the set of lines specified by L1 and L2. After the FIND command is given, the Editor will prompt with:

FIND:

In response, the user should enter the string to be located, terminated by a RETURN. If a RETURN is entered immediately following the prompt, the Editor will return control to the Basic monitor.

FIND will cause each line containing the string to be printed out with a pointer below the line indicating the occurrence of the specified string.

command: **Change String**

format: **CHANGE**

CHANGE L1

CHANGE L1,

CHANGE L1,L2

where:

L1 is an optional line number or name.

If L1 is omitted, all lines of the program are processed.

If L1 is the only argument and there is no comma following it, then L1 is the only line processed.

If L1 is the only argument and the comma is included, L1 through the last line in the program are processed by the command.

L2 is an optional line number or name which indicates the last line to be processed. If included, it must be preceded by L1 and a comma.

The CHANGE command will replace all occurrences of a string within a set of lines specified by L1 and L2. After the CHANGE command is given, the Editor will prompt with:

FROM:

In response, the user should enter the string to be replaced followed by a RETURN. The Editor will then prompt with:

TO:

The user should respond with the replacement string. The Editor will then print the first line containing the string with a pointer below the line indicating the location of the string. The user can then type:

1. A RETURN to reject the CHANGE at that location,
2. The letter (C) to accept the CHANGE, or
3. An asterisk (*) to accept all CHANGEs from that point on.

instruction: **Enter File**

format: **[Ln] ENTER svar**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

svar is a string variable or string literal file reference.

The ENTER instruction is used to ENTER a Basic program (in ASCII format) from a disk file or other external device into the User Area.

Notes:

1. ENTER will read a program which was written to the disk by the LIST instruction. The program will be read into the User Area. ENTER will not read a program which was written to the disk by the SAVE instruction.

2. ENTER does **not** delete statement lines from the program which is currently in the User Area. ENTER **does** replace lines in the current program with lines from the file being ENTERed if the line numbers are the same.

3. Efficient use of memory following an overlay results if the ENTERed program, to as great an extent as is possible, replaces lines in the current program and does not add new line numbers.

4. The ASCII ESCAPE character (1BH) or CONTROL-Z (1AH) are used as the end of file mark by Basic. ENTER looks for either of these characters to determine the end of the program.

The following programs demonstrate the ability of the ENTER instruction to overlay or have one program overlay (part of) itself with another. This feature is very useful for running large programs on smaller systems.

Space is most efficiently used if the same line numbers are used in all programs which are to be overlayed. In the following example, program One is run and calls in program Two.

Program "One.lis"

```
  1   Goto 10
  5   Enter "Two.lis"
 10   Data 1,2,3,4,5,6,7,8,9
 20   Read A,B,C,D,E,F,G,H,I
 30   Goto 5
```

Program "Two.lis"

```
   10    X=A+B+C+D
   20    Y=E+F+G+H+I
   30    Z=X+Y
   40    Print X,Y,Z
   50    End

>>Enter "One.lis"

>>Run
10                      35                      45
***50 End***
```

When program One is entered and RUN, line 1 passes CONTROL to line 10. Then at line 20, the DATA from line 10 is READ into variables A through I. CONTROL is then passed to line 5 which calls in the second program. Program Two overlays lines 10, 20, and 30. The line which follows line 5 is line 10. Now, however, line 10 is a line from program Two. The rest of program Two is executed with the results being printed out by line 40.

instruction: **List Current Program**

format: **[Ln] LIST**

[Ln] LIST L1

[Ln] LIST L1,

[Ln] LIST L1,L2

[Ln] LIST svar

[Ln] LIST svar,L1

[Ln] LIST svar,L1,

[Ln] LIST svar,L1,L2

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

L1 — is an optional line number or line name.

If L1 is omitted, all lines of the program are LISTed.

If L1 is the only argument and there is no comma following it, then L1 is the only line LISTed.

If L1 is the only argument and the comma is included, L1 through the last line in the program are LISTed.

L2 — is an optional line number or line name which indicates the last line to be LISTed. If included, it must be preceded by L1 and a comma.

svar — is a optional string variable or a string literal file reference denoting the pathname of the LISTing. If omitted, the LISTing is displayed on the console.

The LIST instruction is used to LIST in ASCII format one or more statement lines from the User Area on the console, to a disk file, or to another file device. The instruction may be used to output an entire program, a block of statement lines within a program, or a single statement line. The formats of LIST which do not use svar will direct the output to the console.

Notes:

1. A program which has been LISTed to a disk can be read back into the User Area using the ENTER instruction. It can not be read back using LOAD or RUN.

2. LISTed files are compatible between different versions of Cromemco Structured Basic. SAVEd files are not necessarily compatible in this manner.

3. The ASCII ESCAPE character (1BH) is used as the end of file mark by Basic. LIST outputs this character at the end of a program.

instruction: **List Variables**

format: **[Ln] LVAR**

[Ln] LVAR file-ref

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

file-ref is an optional string variable or string literal file-reference denoting the destination of the list of variables. If omitted, the listing is displayed on the console.

The LVAR instruction lists all variables, functions, procedures, and alphanumeric line names. The abbreviations used in the list are:

INT	integer variable
SFP	short floating point variable
LFP	long floating point variable
LBL	alphanumeric label, line name
FUN	function name
PROC	procedure name
$	string variable
(*)	list or matrix

The current value of each scalar arithmetic variable is also displayed.

Notes:

1. If a variable, function, PROCEDURE name or line name is used in a Basic program, and then the line containing that item is deleted, the item will still appear in the list produced by LVAR. The LIST can be updated by LISTing the program to a file, SCRatching the User Area, ENTERing the program, and then using LVAR.

2. If a line name, PROCEDURE name, or function has been defined in the program, its line number will be printed after its type. If a line name, PROCEDURE name, or function has been used (e.g., GOTO Start'over) but not yet defined, no entry will follow the type.

instruction: **Load Program**

format: **[Ln] LOAD svar**

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

svar — is a string variable or string literal file reference.

The LOAD instruction is used to LOAD a Basic program (in internal machine format) from a disk file into the User Area.

Notes:

1. LOAD will read a program which was written to the disk by the SAVE instruction. The program will be read into the User Area. The program must have been SAVEd under the version of Basic and the version of the operating system under which Basic is currently being run.

   Files which are written to the disk by the LIST instruction are compatible between different versions of Cromemco Structured Basic; files written using the SAVE instruction are not.

2. The LOAD instruction resets (clears) all variables, string variables, and matrices (the equivalent of scratch) before LOADing.

3. LOAD resets the trigonometric mode to RADians.

command: **Renumber Statement Lines**

format: **RENUMBER**

RENUMBER L1

L1,L2

RENUMBER L1,L2,L3

RENUMBER L1,L2,L3,

RENUMBER L1,L2,L3,L4

where:

L1 is an optional starting line number in the RENUMBERed program.

L2 is an optional line number increment in the RENUMBERed program.

L3 is an optional line number or line name in the original program.

If L3 is omitted, all lines of the program are RENUMBERed.

If L3 is not followed by a comma then L3 is the only line which is RENUMBERed.

If L3 is followed by a comma, L3 through the last line in the program are RENUMBERed.

L4 is an optional line number or line name which indicates the last line in the original program to be RENUMBERed.

The RENUMBER command alters the statement numbers in the current program.

Notes:

1. The default value for the RENUMBER command is a starting line number (L1) of 10 and an increment value (L2) of 10.

2. If only the first parameter (L1) is specified, the second parameter (L2) assumes the same value as the first. In the example below, the command RENUMBER 100 is equivalent to the command RENUMBER 100,100.

3. The RENUMBER instruction alters line numbers imbedded in the entire program in GOTO, GOSUB, and IF-THEN statements to conform to the RENUMBERed statements. This will affect a line which is not RENUMBERed if the line contains a reference to a RENUMBERed line.

4. RENUMBER cannot normally be used to re-order or rearrange sections of a program relative to other sections. If line numbers are LISTed out of order after the RENUMBER instruction is given, follow the procedure below (5a-d) to rearrange the lines in numeric order.

5. The RENUMBER command will include DELETEd statement numbers in the sequence of RENUMBERed statements. If this presents a problem (such as one or more statement numbers being omitted) the following procedure will correct the problem:

   a. LIST the program (do not SAVE it) to a temporary disk file.

   b. SCRatch the User Area.

   c. ENTER the temporary disk file.

   d. RENUMBER as desired.

Examples:

```
>>List

   11       Input Alpha
   24       Input Beta
   37       Print Alpha*Beta
   50       Goto 11
   63       End

>>Renumber      (default parameters are 10,10)

>>List

   10       Input Alpha
   20       Input Beta
   30       Print Alpha*Beta
   40       Goto 10
   50       End

>>Renumber 100 (if L2 is not specified, L2=L1)

>>List

  100       Input Alpha
  200       Input Beta
  300       Print Alpha*Beta
  400       Goto 100
  500       End

>>Renumber 100,10
```

```
>>List

  100       Input Alpha
  110       Input Beta
  120       Print Alpha*Beta
  130       Goto 100
  140       End

>>Renumber 1000,150,120,
```

(Renumber all lines from line 120 to the end of the program.)

```
>>List

  100       Input Alpha
  110       Input Beta
 1000       Print Alpha*Beta
 1150       Goto 100
 1300       End

>>Renumber 1000,1,110,1150
```

(Renumber lines 110 through 1150 in the current program. The new line numbers will start at 1000 and use an increment of 1.)

```
>>List

  100       Input Alpha
 1000       Input Beta
 1001       Print Alpha*Beta
 1002       Goto 100
 1300       End
```

instruction: **Run Program**

format: **[Ln] RUN**

[Ln] RUN svar

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

svar is a string variable or string literal file pathname.

The RUN instruction directs the computer to execute a program beginning with the lowest numbered line.

If svar is omitted, the program which is currently in partition zero of the User Area is executed.

If svar is included, it must be the name of a SAVEd program. This program will be LOADed into partition zero of the User Area and executed.

Notes:

1. The RUN instruction, if given with a file reference, must reference a program which has been SAVEd under the version of Basic which is currently being used.

   LISTed files are compatible between different versions of Cromemco Structured Basic, SAVEd files are not.

2. The RUN instruction resets or clears all variables, string variables, and matrices. It also sets the trigonometric mode to RADians.

3. If partition zero is not the current partition (after a USE instruction or if execution was terminated in another partition), RUN will cause partition zero to become the current partition and execution to begin in partition zero.

4. RUN resets ON ERROR and ON ESCAPE instructions to their default modes. This means that run-time non-fatal errors as well as the use of the ESCAPE key will cause a running program to abort and Basic to display a message.

5. RUN sets the variable mode to that which was last specified. The default mode is Long Floating Point.

   Refer to the Areas of User Interest Appendix if it is necessary to change the default variable mode.

instruction: **Save Program**

format: **[Ln] SAVE svar**

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

svar — is a string variable or string literal file reference.

The SAVE instruction is used to SAVE the current program on a disk or other file device in internal machine format.

Note:

1. A program which has been written to the disk using the SAVE instruction can be read back using the LOAD or RUN instructions. A SAVEd program can only be LOADed or RUN with the same version of Basic it was SAVEd under.

instruction: **Scratch User Area**

format: **[Ln] SCR**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The SCRatch instruction deletes the current program from the User Area.

Notes:

1. The programmer should keep in mind that the SCRatch command erases everything in all partitions of the User Area and that SCRatched programs cannot be recovered.

2. Once the work space has been SCRatched, the user may input a new program or access a SAVEd or LISTed program which has been stored on the disk.

3. SCRatch sets the trigonometric mode to RADians.

4. SCRatch resets the variable mode to the default mode, normally the Long Floating Point mode.

5. SCRatch does not reset the ECHO or ON ESCAPE mode.

6. SCRatch CLOSEs all OPEN files before performing the actual SCRatch. If any file cannot be CLOSEd (because of disk or I/O problems), some other files may be left OPEN and the User Area will not be SCRatched. The other files may be CLOSEd via the CLOSE\n\ instruction.

Example:

```
>>List

  10          X=4
  20          Input Y
  30          Z=X*2+Y
  40          Print Z
  50          End

>>Scr

>>List

>>
```

In the above example, all statement lines are deleted from memory. The user can now input a new program.

instruction: **Enable Trace Option**

format: **[Ln] TRACE**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The TRACE instruction sets the TRACE mode so that the user can follow the execution of a program line by line. When in the TRACE mode, Basic will list the line number of each statement as it is being executed. Statement line numbers will be enclosed in angle brackets.

Example:

```
>>List

   10          Trace
   20          Input Number
   30          Print"This is ";Number
   40          Let Number1=Number+1
   50          Print Number1
   60          End

>>Run
<20>
? 10
<30>
This is 10
<40>
<50>
11
<60>
***60 End***
```

instruction: **Disable Trace Option**

format: **[Ln] NTRACE**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The NTRACE instruction resets the TRACE mode so that line numbers are not displayed during program execution.

Chapter 9

DOCUMENTATION

This chapter describes the REMark instruction used for documenting programs.

instruction: **Remark**

format: **[Ln] REM text**

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

text — is any string of printable characters.

The REM instruction is used to insert REMarks or comments in a program.

Notes:

1. REM statements included in a Basic program are ignored when the program is executed but are output exactly as entered when the program is LISTed.

2. REM statements occupy space in the User Area. With some long programs, or those with large lists or matrices, it may be necessary to minimize the use of REM statements in order to accommodate the program.

3. Any grammatical or typing mistakes which are made when inputting a REM statement will not generate an error message and will be output precisely as they appear in the statement line.

4. The programmer is encouraged to use REM statements liberally throughout a program to describe program operation. These REMarks can be particularly helpful to any one who wishes to use or modify a program written by another person.

5. Multiple spaces in REMark instructions occupy no more User Area than do single spaces. For this reason, the use of multiple spaces in REMarks is encouraged when it will improve readability.

Chapter 10

ASSIGNMENT INSTRUCTIONS

This chapter describes the LET and MATrix initialization instructions used for assigning values to variables and matrices.

instruction: **Let**

format: **[Ln] LET var = exp**

or

[Ln] var = exp

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

var — is a numeric or string variable or a reference to an element of a matrix.

exp — is the value to be assigned to var. It may be any expression, variable, constant, function, string variable, or string literal.

The LET instruction is used to assign a value to a given numeric variable, string variable, or element of a matrix. The equal sign (=) is called the assignment operator. Refer to chapter 6.

Notes:

1. When a string variable is used with a LET instruction, the portion of the string which is referenced is set equal to null characters before the source is moved into the string variable. The whole string is referenced if no subscripts follow the string variable, while various substrings may be referenced by the use of subscripts. Refer to the section **Referencing String Variables** in Chapter 5 for a complete discussion of the subject. The LET instruction will not move more characters than can be accepted by the destination string or substring which is being referenced.

2. In Basic the equal sign is also used as a relational operator.

3. A string variable or literal may only be assigned to a string variable while any expression or function yielding a numeric result may only be assigned to a numeric variable.

4. Basic is designed to allow the user to assign values to variables without entering LET each time. This capability is called implied LET.

   LET and implied LET instructions execute with equal speed.

Examples:

LET Instructions	**Equivalent Implied LET Instructions**
Let Arm = 45	Arm = 45
Let Bat = Arm + 10	Bat = Arm + 10
Let Text$ = "Basic"	Text$ = "Basic"
Let Total = Part1 + Part2	Total = Part1 + Part2
Let Element = Inversion(10,5)	Element = Inversion(10,5)

instruction: **Matrix Initialization**

format: **MAT mvar = aexp**

where:

aexp is an arithmetic expression, variable, or constant.

mvar is a MATrix arithmetic variable.

The MAT instruction is used to set all elements in a MATrix (M) equal to the value of the arithmetic expression (aexp).

Note:

1. MATrix M must be explicitly dimensioned.

Example:

```
>>List

   10       Dim Array(4)
   20       Read Array(1),Array(2),Array(3),Array(4)
   30       Data 20,21,22,23
   40       Print Array(1),Array(2),Array(3),Array(4)
   50       Mat Array=0
   60       Print Array(1),Array(2),Array(3),Array(4)
   70       Mat Array=1
   80       Print Array(1),Array(2),Array(3),Array(4)
   90       End

>>Run
     20            21            22            23
     0             0             0             0
     1             1             1             1
     ***90 End***
```

Chapter 11

INITIALIZATION

This chapter describes instructions used for defining the numeric variable types. These include instructions to:

-- Define matrices
-- Define for one or all variables whether the mode is integer, long floating point, or short floating point
-- Define whether the mode for trigonometric calculations is degrees or radians

MIXED MODE OPERATIONS

There are three types of numbers in Basic, INTEGER, Short Floating Point (SHORT), and Long Floating Point (LONG), (hexadecimal constants are treated as integers). Any arithmetic computation, assignment, INPUT, or READ can be performed using any one or more of these types of numbers.

An assignment, READ, or INPUT will automatically convert the number to the type of the receiving variable. This is the variable which is on the left of the equal sign in an assignment instruction, and in the DATA list in the READ and INPUT instructions.

In general, numbers are converted to other types freely as needed. For example:

Sin(30) works, even though the SIN function must have a LONG argument. The Integer 30 is converted to a Long Floating Point number.

Sys(2.6) works, even though the SYS function must have an INTEGER argument. The Long (or Short) Floating Point number 2.6 is converted to an integer (rounding to 3.0).

Most problems will occur with mixed mode arithmetic involving INTEGER numbers. Remember that all constants without a decimal point and with a value less than 10,000 are stored as INTEGERs.

Examples:

```
>>Short Short'var
>>Integer Integer'var
>>Short'var = 6 : Integer'var = 1
>>Short'var = Integer'var / 3 * Short'var
```

This example will assign a value of 0 to Short'var. This is because Integer'var/3 is evaluated first (rules of precedence, arithmetic operators). Because both Integer'var and 3 are INTEGERs, Integer'var/3 is evaluated as a 0 using integer arithmetic. Zero times anything, no matter what type, is still 0.

```
>>Short Short'var
>>Integer Integer'var
>>Short'var = 6 : Integer'var = 1
>>Short'var = Short'var * Integer'var / 3
```

This example will assign a value of 2 to Short'var. This time Short'var*Integer'var is evaluated first, and because this is mixed mode arithmetic, the shorter form is converted to the longer form (the value of Integer'var is converted to SHORT). Then Short'var*Integer'var is equivalent to 6.0*1.0 or 6.0. We are left with 6.0/3, mixed mode again.

The Integer 3 (the shorter type) is converted to Short Floating Point (the longer type) and the division is performed. Short'var is assigned a value of 6.0/3.0 or 2.0.

```
>>Integer Integer'var
>>Short Short'var
>>Long Long'var
>>Integer'var = 1 : Short'var = 3 : Long'var = 1
```

If at this point we give the command:

```
>>Long'var = Long'var/Short'var
```

Long'var will be assigned a value of 0.33333333333333 because Long'var/Short'var is evaluated as Long Floating Point (the longer type).

If instead we had given the command:

```
>>Long'var = Integer'var/Short'var
```

Long'var would have been assigned the value of 0.33333300000000 because I/S is evaluated as Short Floating Point (the longer type, but still only 6 digits of accuracy) and then assigned to a Long Floating Point variable (Long'var) with 14 digits of accuracy.

Conversion of both types of floating point numbers to Integer numbers and vice versa does take time. Also arithmetic, indexing, and subscripting with floating point numbers is much more time consuming than it is using INTEGERs.

Time can be saved by using INTEGER numbers wherever possible. Where it is not possible, care in precedence ordering can result in significant time savings. If Long'var is type LONG and Integer'var is type INTEGER, the first of the following commands will execute faster than the second one.

```
>>Long'var = Integer'var*Integer'var*Integer'var*Long'var
>>Long'var = Long'var*Integer'var*Integer'var*Integer'var
```

This is because, until the last multiplication, the first example is using INTEGER arithmetic. The second example uses Long Floating Point arithmetic from the start, because the Long variable is at the left and this is where evaluation of this expression begins.

instruction: **Degree Mode**

format: **[Ln] DEG**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The DEG instruction sets the trigonometric calculation mode to DEGree.

Note:

1. RUN, SCRatch, and LOAD will automatically reset the trigonometric calculation mode to RADian.

instruction: **Dimension**

format: **[Ln] DIM svar(aexp1)**

[Ln] DIM avar(aexp1)

[Ln] DIM avar(aexp1,aexp2)

[Ln] DIM avar(aexp1,aexp2,aexp3)

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
avar	is a matrix arithmetic variable.
svar	is a string variable.
aexp1-3	are arithmetic expressions, variables or constants.

The DIM instruction is used to define the size of a matrix or a string variable. Cromemco Basic permits the user to define one, two, or three DIMensional matrices.

Notes:

1. A DIMensioned numeric matrix variable can have the same name as any other numeric variable. A DIMensioned string variable must have the name of a string variable.

2. If a matrix or string variable is not explicitly dimensioned in a program, the default value of 10 (11 elements, numbered 0 through 10) will be automatically assigned to a singly subscripted matrix or string variable.

   Doubly and triply subscripted matrices will generate an error message if not explicitly dimensioned.

3. The maximum size of any matrix is only restricted by the amount of available memory. Any single DIMension may not exceed 16,382.

4. The DIMension of a string variable may not exceed 32766.

5. The first element in a matrix is numbered 0 (zero indexing).

instruction: **Integer Mode**

format: **[Ln] IMODE**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The IMODE instruction changes the default mode of all variables to the Integer mode.

Notes:

1. This instruction takes effect only after the execution of a RUN instruction after the IMODE instruction has been given. See the following example.

2. Integer variables occupy 2 bytes and must be within the range +32767 to -32768.

3. This instruction will be overridden by the LONG and SHORT instructions.

Example:

```
1 Imode : X=2.5 : If X=2.5 Then Run
```

This line, appearing as the first line of a program, will ensure that the interpreter is in the Integer mode. If the line is encountered while the interpreter is not in the Integer mode, the IMODE instruction will be given, X will be set equal to 2.5 (a non integer number), and if X=2.5 (as will be the case if SFMODE or LFMODE are current) the RUN instruction will be executed. Program execution will then begin over again, this time the RUN instruction will be given after the IMODE instruction and the current mode will be integer. When the value 2.5 is assigned to X, X will be an integer variable so that 2.5 will be rounded and X will have the value of 3. Then X=2.5 will be false and CONTROL will be transferred to the next line.

instruction: **Integer Variable**

format: **[Ln] INTEGER avar, mvar(x),...**

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
avar	is a scalar arithmetic variable
mvar	is a matrix arithmetic variable.
x	is the optional DIMension of mvar.

The INTEGER instruction is used to set a given variable to the INTEGER mode.

Notes:

1. DIMensioning may be done via the INTEGER instruction.
2. The INTEGER instruction must be executed before the variable is referenced for the first time.
3. INTEGER variables occupy 2 bytes and must be within the range +32767 to -32768.
4. This instruction overrides the SFMODE and LFMODE instructions.

instruction: **Long Floating Point Mode**

format: **[Ln] LFMODE**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The LFMODE instruction is used to set all variables within a program to the Long Floating Point mode.

Notes:

1. This is the standard default mode for all variables and arithmetic operations.

2. This instruction takes effect only after the execution of a RUN instruction after the LFMODE instruction has been given. See the following example.

3. Long Floating Point variables occupy 8 bytes and must be within the range ±9.99E+62 to ±9.99E-65. They have an accuracy of 14 digits.

4. This instruction will be overridden by the INTEGER and SHORT instructions.

Example:

```
1 Lfmode : X=0.12345678 : If X<>0.12345678 Then Run
```

This line, appearing as the first line of a program, will ensure that the interpreter is in the Long Floating Point mode. If the line is encountered while the interpreter is not in the Long Floating Point mode, the LFMODE instruction will be given, X will be set equal to 0.12345678 (a number that cannot be represented in either Short Floating Point or Integer modes), and if X does not equal 0.12345678 (as will be the case if SFMODE or IMODE are current), the RUN instruction will be executed. Program execution will then begin over again, this time the RUN instruction will be given after the LFMODE instruction and the current mode will be Long Floating Point. When the value 0.12345678 is assigned to X, X will be a Long Floating Point variable with the value of 0.12345678. Then X<>0.12345678 will be false and CONTROL will be transferred to the next line.

instruction: **Long Variable**

format: **[Ln] LONG avar, mvar(x),...**

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
avar	is a scalar arithmetic variable.
mvar	is a matrix arithmetic variable.
x	is the optional DIMension of mvar.

The LONG instruction is used to set a given variable to the Long Floating Point mode.

Notes:

1. DIMensioning may be done via the LONG instruction.
2. The LONG instruction must be executed before the variable is referenced for the first time.
3. Long Floating Point variables occupy 8 bytes, have an accuracy of 14
4. SCRatch, RUN, and LOAD will automatically reset the trigonometric mode to RADian.

instruction: **Radian Mode**

format: **[Ln] RAD**

where:

[Ln] is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise, it is executed immediately.

The RAD instruction sets the RADian mode for trigonometric calculations.

Note:

1. SCRatch, RUN, and LOAD will automatically reset the trigonometric mode to RADian.

instruction: **Short Floating Point Mode**

format: **[Ln] SFMODE**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The SFMODE instruction is used to set all variables within a program to the Short Floating Point mode.

Notes:

1. This instruction takes effect only after the execution of a RUN instruction after the SFMODE instruction has been given. See the following example.

2. Short Floating Point variables occupy 4 bytes, have an accuracy of 6 digits, and must be within the range ±9.99E+62 to ±9.99E-65.

3. This instruction will be overridden by the LONG and INTEGER instructions.

Example:

```
1 Sfmode : X=0.90000001 : If X<>0.9 Then Run
```

This line, appearing as the first line of a program, will ensure that the interpreter is in the Short Floating Point mode. If the line is encountered while the interpreter is not in the Short Floating Point mode, the SFMODE instruction will be given, X will be set equal to 0.90000001 and if X is not equal to 0.9 (as will be the case if IMODE or LFMODE are current) the RUN instruction will be executed. Program execution will then begin over again, this time the RUN instruction will have been given after the SFMODE instruction and the current mode will be Short Floating Point. When the value 0.90000001 is assigned to X, X will be a Short Floating Point variable so that 0.90000001 will be rounded and X will have the value of 0.9. Then X<>0.9 will be false and CONTROL will be transferred to the next line.

instruction: **Short Variable**

format: **[Ln] SHORT avar, mvar(x),...**

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
avar	is a scalar arithmetic variable.
mvar	is a matrix arithmetic variable.
x	is the optional DIMension of mvar.

The Short instruction is used to set a given variable to the Short Floating Point mode.

Notes:

1. DIMensioning may be done via the SHORT instruction.
2. The SHORT instruction must be executed before the variable is referenced for the first time.
3. Short Floating Point variables occupy 4 bytes, have an accuracy of 6 digits, and must be within the range ±9.99+62 to ±9.99E-65.
4. This instruction overrides the IMODE and LFMODE instructions.

Chapter 12

CONTROL STRUCTURES

This chapter describes the instructions used for controlling the flow of execution within a program. These include instructions to:

-- Set up loops
-- Call subroutines
-- Go to specific lines
-- Stop and end execution

command: **Continue Program Execution**

format: **CON**

The CON command CONtinues program execution after a program is interrupted by a STOP statement, a program error, or the ESCAPE key.

Notes:

1. Program execution will commence with the line following the statement at which the program stopped.

2. If program execution stopped because of a program error, the error can be corrected and the CON command used to continue execution from the line following the one where the error occurred.

3. If the user wishes to re-execute the line in error, GOTO (as a command) should be used.

statement: **End Program Execution**

format: **Ln END**

where:

Ln is a line number.

The END statement halts program execution and causes Basic to return to the command mode.

Notes:

1. Unlike the STOP statement, program execution may not be CONtinued after an END statement has been executed.

2. Upon execution of the END statement, Basic displays a message on the console indicating the line number of the END statement which caused the program to halt.

Example:

```
>>List

  100   Print "Tomorrow and tomorrow and tomorrow..."
  200   End

>>Run

Tomorrow and tomorrow and tomorrow...
***200 End***

>>
```

instructions: **For-Next Loop**

format:
```
[Ln] FOR avar=aexp1 To aexp2 [Step aexp3]
      .
      .
[program instructions]
      .
      .
[Ln] NEXT avar
```

where:

Ln — are optional line numbers. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

avar — is a non-subscripted numeric variable. This is the index variable.

aexp1-3 — are arithmetic expressions, variables, or constants. They may not be string literals or string variables. These are the FOR-NEXT loop parameters.

aexp1 is the initial value.
aexp2 is the final value.
aexp3 is the step value.

The FOR-NEXT instructions are used to repeat a part of a Basic program a specified number of times. During the execution of a FOR-NEXT loop, an index (avar) is maintained. When this index becomes equal to (or greater than, or less than) the final value (aexp-3), CONTROL is transferred to the instruction following the NEXT instruction.

Notes:

1. The following sequence defines the execution of a FOR-NEXT loop:

   a. The expressions aexp1, aexp2, and aexp3 are evaluated. If aexp3 is omitted it is given a value of +1 (if the step value is not specified it is assumed to be +1).

   b. The index variable (avar) is set equal to the initial value (aexp1).

   c. The instructions following the FOR and preceding the NEXT are executed.

   d. The step value is added to the index variable (avar = avar + aexp3).

e. If the step value (aexp3) is positive and the index variable (avar) is greater than or equal to the final value (aexp2), then the condition for termination of the FOR-NEXT loop has been met and CONTROL is transferred to the instruction following the corresponding NEXT instruction.

   If the step value (aexp3) is negative and the index variable (avar) is less than or equal to the final value (aexp2), then the condition for termination of the FOR-NEXT loop has been met and CONTROL is transferred to the instruction following the corresponding NEXT instruction.

   Otherwise execution of the loop continues with C above.

2. The step (aexp3) portion of the FOR instruction is optional. If a step value is not specified, Basic assumes a value of +1 for aexp3.

3. Basic programs will execute significantly faster if the loop parameter variables are declared as type Integer.

4. FOR-NEXT loops may be nested within a program. It is important to keep in mind, however, that each FOR instruction and its corresponding NEXT instruction must be completely contained within any larger loop.

5. Program LISTings have FOR-NEXT loops indented for clarity.

Examples:

```
>>List


   10      Rem Demonstration Program
   20          For Index'var = 0 TO 10 Step 2
   30          Sum = Index'var+1
   40          Print Sum; " "
   50          Next Index'var
   60       End

>>Run
1357911***60End***
```

In the above example, upon entering the FOR-NEXT loop defined by lines 20 through 50, the loop parameters are evaluated and set equal to:

```
 0 (aexp1, the initial value)
10 (aexp2, the final value)
 2 (aexp3, the step value)
```

The index variable is then set equal to the initial value (Index'var=0). Execution continues with lines 30 and 40 where the Index'var maintains the value assigned to it by the FOR instruction. At line 50 the index variable is tested to see if it is greater than or equal to the final value (is Index'var > 10?). In this case 0 is not greater than 10, so execution continues with line 20. The step value is added to the index variable (Index'var = Index'var + aexp3). This continues until (Index'var >=aexp2). Then CONTROL is transferred to statement 60.

Correct Nesting Format:

```
 10     For Var1 = 1 TO 50 Step 2
 20       For Var2 = 1 TO 30 Step 5
 30         For Var3 = 1 TO 10 Step 1
          .
          .
          .
100         Next Var 3
110       Next Var2
120     Next Var1
```

Illegal nesting occurs when FOR-NEXT loops overlap. The following example will generate a run-time error.

Incorrect Nesting Format

```
 10     For Var1 = 1 TO 50 Step 2
 20       For Var2 = 1 TO 30 Step 5
 30         For Var3 = 1 TO 10 Step 1
          .
          .
          .
100         Next Var2
110       Next Var3
120     Next Var1
```

instruction: **Gosub-Return**

format:
```
[Ln] GOSUB n
      .
      .
n [program instructions]
      .
      .
[Ln] RETURN
```

where:

Ln — are optional line numbers. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n — is the line number or line name of the first statement of the subroutine to which CONTROL is transferred.

The GOSUB instruction transfers CONTROL to a subroutine.

When, during the execution of the subroutine, a RETURN instruction is executed, CONTROL is passed to the instruction following the GOSUB instruction which called the subroutine.

Notes:

1. In Basic, subroutines may be fully enclosed or nested in other subroutines.

   When nesting subroutines, remember that the RETURN instruction will take you back to the last GOSUB and start execution of the instruction immediately following. Improperly nested GOSUB-RETURN instructions will generate run time error messages.

   A nested subroutine is executed after the GOSUB statement and before the RETURN statement of the subroutine in which it is enclosed.

Example:

```
>>List

   10      Rem Example program, GOSUB
   20      Gosub Demo'print
   30      Print "Program Over"
   40      End
   50      *Demo'print : Print "This is a subroutine"
   60      Print "which demonstrates the"
   70      Print "Gosub statement"
   80      Print
   90      Return

>>Run
This is a subroutine
which demonstrates the
Gosub statement

Program Over
***40 End***
```

instruction: **Gosub-Retry**

format: **[Ln] GOSUB n**

.
.

n [program instructions]

.
.

[Ln] RETRY

where:

Ln — are optional line numbers. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n — is the line number or line name of the first statement of the subroutine to which CONTROL is transferred.

The GOSUB instruction transfers CONTROL to a subroutine.

The RETRY instruction is similar to the RETURN from a subroutine instruction except that it re-executes the instruction which called the subroutine.

Notes:

1. RETRY is to be used in conjunction with the ON ERROR-GOSUB error trapping instruction only.

   When used with an error trap, the condition which caused the error can be fixed and the statement which caused the error will be re-executed.

2. When used in place of RETURN in a subroutine which was called by a standard GOSUB instruction, RETRY will cause Basic to repeatedly execute the subroutine without end.

Example:

```
100  Procedure .Open'file (File'name$)
110    Rem
120    Rem This Procedure OPENs a file on channel 1.
130    Rem If the file does not exist it will be CREATEd
140    Rem before it is OPENed.
150    Rem
160    On Error Gosub File'create :  Rem Set error trap
170    Open\1\File'name$
180    On Error Stop :  Rem Reset error trap.
190    Endproc
200    Rem
210    Rem
220    Rem Subroutine FILE'CREATE
230    Rem If called because of an error 134
240    Rem (Cannot Open File) this routine will CREATE
250    Rem the file.  If called because of any other
260    Rem error, the Basic error flag will be set,
270    Rem the error trap will be reset, and the
280    Rem Procedure aborted.
290    Rem
300  *File'create
310    Error'number=Sys(3)
320    If Error'number=134 Then Do
330      Create File'name$
340      Else
350      On Error Stop
360      Errproc
370      Enddo
380    Retry
390    End
```

instruction: **Goto**

format: **[Ln] GOTO n**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n is the line number or line name line number of the statement to which CONTROL is transferred.

The GOTO instruction unconditionally transfers CONTROL to the statement line specified by n.

When used as a statement, GOTO interrupts the normal execution sequence of program statements and transfers CONTROL to the specified statement.

When used as a command GOTO will cause execution of a program to start with the specified statement.

Notes:

1. When used with the IF-THEN instruction and a line number, the words GOTO are optional:

```
If Bool=0 Then Goto 50

   or

If Bool=0 Then 50
```

   are equivalent and are both legal instructions.

2. When execution of a program is initiated or continued using a GOTO command, no variable initialization takes place.

Example:

```
>>List

   10      Input Number
   20      If Number<0 Then 200
   30      Root = Sqr(Number)
   40      Print "The square root of ";Number;" is ";Root
   50      Goto 210
  200      Print "This yields an imaginary number"
  210      End

>>Run
? -2
This yields an imaginary number
***210 End***
>>Run
? 9
The square root of 9 is 3
***210 End***
```

In statement 20 the words GOTO are omitted. This statement causes CONTROL to be transferred to statement 200 if the condition (A<0) is true. Statement 50 unconditionally transfers CONTROL to statement 210.

instruction: **If-Then**

format-1: **[Ln] IF exp THEN n**

format-2: **[Ln] IF exp THEN instruction**

[Ln] IF exp THEN instruction:instruction:...

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

exp — is a relational or arithmetic expression, an arithmetic variable, or a constant.

n — is the line number or line name of the statement to which CONTROL is transferred.

instruction — is any Basic instruction except FOR, NEXT, REM or DATA.

The IF-THEN instruction evaluates exp to false (=0) or true (not equal to 0).

If format-1 is used, CONTROL is transferred to the specified statement if exp is evaluated as true, and to the next sequential statement if exp is false.

If format-2 is used, and if exp is true, the instructions remaining on the same line are executed before CONTROL is transferred to the next sequential statement. If exp is false, CONTROL passes to the next sequential statement line.

Notes:

1. In a relational expression, a relational operator (=, <, >=, <=, <>, or >) is used to compare two expressions or values. Refer to Chapter 6 for a discussion of relational and Boolean operators.

2. No instruction may follow format-1 of the IF-THEN instruction on the same line.

3. IF-THEN instructions may be nested using format-2.

   For example:

   ```
   100  If A = B Then If G$ = "N" Then 500
   ```
   is a legal statement.

4. Also refer to the IF-THEN-ELSE instructions.

Examples:

```
100 *In : Input Part'number
110   If Part'number=0 Then @"must be non-zero" : Goto In
120   Print Part'number
130   End
```

In this example, the computer outputs a prompt (?) to which the user responds with a number. IF the number is non-zero (Part'number=0 is false), CONTROL will be passed to line 120, the number will be printed, and execution of the program will terminate. IF the number is zero (Part'number=0 is true), the part of line 110 after THEN will be executed, printing out the message and returning CONTROL to the line labeled In (line 100) which will request another number from the user.

```
100 *Again : Input "First Name: ", Name$
200   If Name$ = "FRED" Or Name$ = "Fred" Then Goto Finish
300   Goto Again
400 *Finish : Print "So you are Fred!"
500   End
```

This program will continue to prompt the user with "First Name:" until the user responds with either Fred or FRED. The IF-THEN instruction includes a compound relational expression:

```
Name$ = "FRED" Or Name$ ="Fred"
```

This expression is evaluated as true (=1) if either FRED (all upper case) OR Fred (Upper case F, lower case red) is entered. This type of checking is very useful in interactive programs where a variety of user responses are to be allowed.

statements: **If-Then-Else**

format:
```
Ln IF exp THEN DO
      .
      .
      .
[Ln ELSE]
      .
      .
      .
Ln ENDDO
```

where:

Ln	are line numbers
exp	is a relational or arithmetic expression.

If exp evaluates to true (not equal to zero) then the program instructions following DO are executed. If, in addition, ELSE is coded, ELSE will transfer CONTROL to the ENDDO statement.

If exp evaluates to false (equal to zero) then CONTROL is transferred to ELSE (if ELSE is used) or to ENDDO (if ELSE is not used).

Example:

The following program will request a line of text from the user, and then count the number of words and non-blank characters in the line. The average number of characters per word will be computed and displayed and then the user will be prompted for another line of text. Entering a RETURN in response to the request for INPUT will terminate the program.

```
100  Dim Buffer$(100)
110  Dim Blank$(0) :  Blank$=" "
120  Rem initialize blank character flag
130  Rem word counter
140  Rem character counter
150  Last'char'was'blank=0
160  Number'of'words=1
170  Number'of'char=0
180  Rem prompt user
190  Input"Enter line:  ",Buffer$
200  Rem check for user termination (null buffer)
210  Length'of'buffer=Len(Buffer$)
220  If Length'of'buffer=0 Then Goto 510
230  Rem
240  Rem
250  Rem loop through buffer
260    For Index=0 To Length'of'buffer-1
270    Rem check for blank character
280    If Buffer$(Index,-1)=Blank$Then Do
290      Rem check if the last character was a blank
300      If Last'char'was'blank=0 Then Do
310        Rem if not, increment word counter and
320        Rem set flag
330        Number'of'words=Number'of'words+1
340        Last'char'was'blank=1
350        Enddo
360      Rem if not a blank, increment character counter
370      Rem and reset flag
380      Else
390      Number'of'char=Number'of'char+1
400      Last'char'was'blank=0
410      Enddo
420    Next Index
430  Rem
440  Rem
450  Rem display results and return for another user
     entry
460  @"Number of non-blank characters is
     ";Number'of'char
470  @"Number of words is ";Number'of'words
480  Avg=Number'of'char/Number'of'words
490  @"Average number of characters/word is ";Avg
500  Goto 150
510  End
```

instructions: **On-Goto**
On-Gosub

format: **[Ln] ON aexp GOTO n1,n2,...,ni**

[Ln] ON aexp GOSUB n1,n2,...,ni

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

aexp — is an arithmetic expression, variable, or constant

n1-i — are line numbers and/or line names of the statements to which CONTROL can be transferred.

The ON-GOTO and ON-GOSUB instructions transfer CONTROL to any one of several lines in a program based on the value of the expression (aexp) contained in the instruction.

Refer to the GOTO and GOSUB instructions for more information.

Notes:

1. When aexp is evaluated, if it is equal to one, CONTROL will be passed to the statement n1; if it is equal to two, CONTROL will be passed to the statement n2; if it is equal to i, CONTROL will be passed to the statement ni.

2. If aexp evaluates to a non integer number the value of the number will be rounded to the nearest integer for the purpose of these instructions.

3. If aexp is evaluated as less than one or greater than i the instruction will be ignored and CONTROL will pass to the next sequential instruction.

4. If n1, n2, or ni is a nonexistent line, and if CONTROL is transferred to that line, a fatal run time error will be generated.

5. No instruction may follow an ON-GOTO or ON-GOSUB instruction on the same line.

statements: **Repeat-Until Loop**

format: **Ln REPEAT**

.
.
.

Ln UNTIL exp

where:

Ln is a line number.

exp is an arithmetic or relational expression.

The REPEAT structure is used to REPEAT a set of Basic program instructions UNTIL exp is evaluated as true (not equal to zero).

Note:

1. The Basic program instructions contained within the REPEAT-UNTIL structure will be executed at least one time. If exp is true when it is evaluated, CONTROL is passed to the instruction following the UNTIL instruction. Otherwise program CONTROL is passed back to the REPEAT instruction.

Example:

```
1000   Input"Number, limit:  ",Number,Limit
1010   Power'of'number=Number
1020     Repeat
1030     Print Power'of'number
1040     Power'of'number=Power'of'number*Number
1050     Until Power'of'number>=Limit
1060   End
```

This program will prompt the user for a number and a limit. The series of powers of the number will be displayed until the limit is reached.

statements: **While-Endwhile Loop**

format:
```
Ln WHILE exp
      .
      .
[program instructions]
      .
      .
Ln ENDWHILE
```

where:

Ln — are line numbers.

exp — is an arithmetic or relational expression.

The WHILE structure is used to repeatedly execute a section of a Basic program WHILE a condition is true.

Notes:

1. If exp is false when it is evaluated, CONTROL passes to the instruction following the ENDWHILE instruction. Otherwise program CONTROL continues with the next sequential instruction.

2. It is possible for the code contained in the WHILE structure not to be executed at all. This will happen if exp is false the first time the WHILE instruction is executed.

Example:

```
110  Integer Valid'answer'flag,True,False
120  True=1 :  False=0
130  Valid'answer'flag=False
140    While Not Valid'answer'flag
150    Input"Answer(y/n):  ",Answer$
160    If Answer$="Y"Or Answer$="y"Then Valid'answer'flag=True
170    If Answer$="N"Or Answer$="n"Then Valid'answer'flag=True
180    If Not Valid'answer'flag Then Print"Invalid response.  ";
190    Endwhile
200  Rem The program continues here,
210  Rem once a valid answer has been entered.
```

```
>>run
Answer(y/n):  57
Invalid response.  Answer(y/n):  new
Invalid response.  Answer(y/n):  Y

***END***
```

This section of code demonstrates the use of the WHILE structure. The WHILE condition (Not Valid'answer'flag) becomes false, and the program continues, only when the user provides an acceptable response.

The first program line defines the variables (Valid'answer'flag, True, and False) as type INTEGER. Since all Basic operations involving Integer variables are performed faster than those involving Floating Point variables, defining these variables as type INTEGER allows for faster program execution. Note that throughout the program, these are used strictly as Boolean-type variables. This means that they may only take on the values of true (=1) or false (=0). On the second line, the variables True and False are initialized.

On line 130, the variable Valid'answer'flag is set to False (=0) because, before the user is prompted, no valid answer has been accepted by the program.

Line 140 defines the condition for execution of the WHILE loop. This statement line can be read as, "WHILE the variable Valid'answer'flag is not true, perform the instructions up to the ENDWHILE instruction and then return CONTROL to this test (line 140)". If the portion of the statement following the WHILE is true, CONTROL remains within the WHILE structure. If it is false, CONTROL is transferred to the instruction following the ENDWHILE. In this example, the boolean operator NOT causes CONTROL to remain within the WHILE loop as long as Valid'answer'flag is NOT true. When a valid answer is INPUT by the user, Valid'answer'flag is set to true, the WHILE test condition becomes false, and CONTROL passes to the instruction following the ENDWHILE.

Lines 150 through 180 prompt the user for a response and determine if the response meets the criteria of the programmer. If a valid response is INPUT, the value of the variable Valid'answer'flag is set to true (=1), otherwise it remains as false (=0). If Valid'answer'flag is false, an error message is displayed before the user is reprompted.

Upon encountering the ENDWHILE statement, CONTROL is transferred to the WHILE statement where a test is performed as indicated above.

statement: **Stop Program Execution**

format: **Ln STOP**

where:

Ln is a line number.

The STOP statement halts program execution and causes Basic to return to the command mode.

Notes:

1. After a program has been STOPped by a STOP statement execution may be restarted from the statement line immediately following the line containing the STOP statement by the use of the CONtinue command.

2. Upon execution of the STOP statement, Basic displays a message on the console indicating the line number of the STOP statement which caused the program to halt.

Example:

```
>>List

   10   Input A,B,C,D,E,F
   20   Let Number=A+B/C
   30   Print A;B;C
   40   Stop
   50   Print D;E;F
   60   Print Number
   70   End

>>Run
? 1,2,3,4,5,6
123
***40 Stop***
>>Con
456
1.6666666666667
***70 End***
```

In this example, the Basic returns to command mode after encountering the STOP in statement line 40. The program is then CONtinued when the user enters the CON command.

Chapter 13

CONSOLE AND DATA INPUT/OUTPUT

This chapter describes the instructions used for reading and writing data to and from the console and for setting up DATA statements.

instruction: **Input (from the console)**

format: **[Ln] INPUT var-1,var-2,...,var-n**

[Ln] INPUT "string", var-1,var-2,...,var-n

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
var1-n	is a list of one or more numeric and/or string variables.
string	is an optional string literal. It is a prompt to be displayed on the console.

The INPUT instruction assigns a value, INPUT from the console, to a variable. When the INPUT instruction is executed, a prompt (either a question mark (?) or a string as used above) is output to the console. The user should respond to this prompt by entering a list of data which corresponds to the variable list in the INPUT instruction. The data list entered by the user must be terminated by a RETURN.

Notes:

1. If the string format of the INPUT instruction is used, the string will replace the question mark as the initial prompt.

2. If the user types in fewer data items than are called for in the variable list, a double question mark will appear on the terminal. This prompt continues to appear until each variable has been assigned a value. See the examples.

3. If more data is INPUT than is required, an error message will be generated.

4. The type of data INPUT must be the same as the type of variable listed in the INPUT instruction. For example, if the INPUT command contains a list of numeric variables, the data INPUT must be numeric data. If an attempt is made to INPUT string data into a numeric variable, an error message will be generated.

5. When a string variable is used with an INPUT instruction, the portion of the string which is referenced is set equal to null characters before the source is moved into the string variable. The whole string is referenced if no subscripts follow the string variable, while various substrings may be referenced by the use of subscripts. Refer to the section **Referencing String Variables** in Chapter 5, for a complete discussion of the subject. The INPUT instruction will not move more characters than can be accepted by the destination string (or substring) which is being referenced.

6. If the INPUT list is terminated with a semicolon, the RETURN which the user types after the requested data has been entered will not be echoed. This allows more than one prompt and response sequence to appear on a single line.

7. Refer also to the description of the INPUT instruction in the chapter on Data File I/O.

Examples:

```
>>List

   10   Input Num1,Num2,Num3,Num4
   20   Print Num1
   30   Input Num5,Num6
   40   Print Num2;" ";Num3;" ";Num4;" ";Num5;" ";Num6

>>Run
? 32,18,20,4
32
? 100,200
18 20 4 100 200
***50 End***
```

In the above example, the proper number of items was INPUT in response to each prompt. Suppose we were to respond to INPUT statement 10 with only three numbers:

```
>>Run
? 5,10,15
?? 20
5
? 30,40
10 15 20 30 40
***50 End***
```

Basic displayed an additional prompt (??) when it did not receive as many items as were in the INPUT list.

In our example, line 10 and line 30 can be replaced with lines which will tell the user specifically what items are needed:

```
>>10 Input "Enter four numbers:  ",Num1,Num2,Num3,Num4
>>30 Input "The last two numbers: ",Num6,Num7
>>Run
Enter four numbers:  100,200,300,400
100
The last two numbers:  500,600
200 300 400 500 600
***50 End***
```

instruction: **Print (to the console)**

format: **[Ln] PRINT**

[Ln] PRINT A1,A2,...,An

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

A1-n — are numbers, string literals, numeric variables, string variables, numeric expressions, or standard or user defined functions.

The PRINT instruction is used to output information to the terminal.

Notes:

1. When used alone, the PRINT instruction will generate a new line (RETURN, LINE FEED).

2. The at sign (@) may be used in place of the word PRINT for brevity.

Example:

```
30 Print
40 Print "And it saves space"
```

can also be written as:

```
30 @
40 @ "And it saves space"
```

This feature facilitates entry of large, text-oriented programs.

3. The spacing of output can be controlled by using a comma (,) or a semicolon (;) between items in a PRINT list.

Items separated by commas are PRINTed beginning in the leftmost column of each PRINT field. Using the default value of 20 columns per field, the PRINT statement below:

```
10 Print 1,2,3,4
```

will produce output in the following format:

	1	2	3	4
column no.	0	20	40	60

See the SET instruction to change the number of columns per field.

If a semicolon is used between items, the items are printed adjacent to each other (i.e., without spaces between items). For example, the statement:

```
10 Print "Al"; "to"; "ge"; "th"; "er"
```

will produce output in the following format:

```
Altogether
```

Commas and semicolons may be used in a PRINT instruction in any combination.

4. If more items are listed in a PRINT statement than can be output on one line, Basic will generate a LINE FEED and continue PRINTing on the next line.

5. More complex formatting of PRINTed text is possible with the TAB and SPC functions and the PRINT USING instruction.

6. Refer also to the description of the PRINT instruction in the chapter on Data File I/O.

Example:

```
>>List

   20      Current'year=1979
   30      Print "This is ";Current'year
   40      Print
   50      Print
   60      Input Birth'date
   70      Age=Current'year-Birth'date
   80      Print "You were born in ";Birth'date
   90      Print
  100      Print "You are "; Age;" years old this year"
  110      End

>>Run
This is 1979


? 1952
You were born in 1952

You are 27 years old this year
***110 End***
```

instruction: **Read Data**

format: **[Ln] READ var-1,var-2,...,var-n**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

var1-n is a list of one or more numeric and/or string variables

The READ instruction is used to READ values from DATA statement(s) and assign these values to the READ instruction variable list.

Notes:

1. The order in which variables appear in the READ instruction determines which value from the DATA list will be assigned to which variable. For instance, the first value that appears in the DATA list is assigned to the first variable in the READ list, the fifth value is assigned to the fifth variable, and so forth.

2. A pointer is moved in sequence through the list of DATA values as these values are assigned to variables in the READ list. The number of DATA elements must be equal to or greater than the number of variables in the READ list. If there are fewer items remaining in the DATA list than are in the current READ list, an error message will be generated.

3. DATA elements corresponding to numeric variables must be numeric DATA and elements corresponding to string variables must be string literals.

4. When a string variable is used with a READ instruction, the portion of the string which is referenced is set equal to null characters before the source is moved into the string variable. The whole string is referenced if no subscripts follow the string variable, while various substrings may be referenced by the use of subscripts. Refer to the section **Referencing String Variables** in Chapter 5, for a complete discussion of the subject. The READ instruction will not move more characters than can be accepted by the destination string (or substring) which is being referenced.

Example:

```
>>List

   10   Read Bananas,Pears,Peaches,Company$
   20   Print Bananas,Pears,Peaches,Company$
   30   Data 10,20,30,"Fruit Co."
   40   End

>>Run
10            20            30            Fruit Co.
***40 End***
```

instruction: **Restore Data Pointer**

format: **[Ln] RESTORE**

[Ln] RESTORE n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n is a line number or line name in the current program.

The RESTORE instruction resets the DATA list pointer which is associated with the READ instruction. This allows the user to reREAD or skip over DATA items.

Note:

1. Line n must be a line of or before a DATA statement. If n is not used, the DATA list pointer is positioned before the first DATA list item in the program. If n is included, the DATA list pointer is positioned before the first occurring DATA list item on or after the specified line.

Example:

```
>>List

  10      Read A,B,C,D
  20      Read E,F,G,H
  30      Restore
  40      Read R,S,T,U
  50      Restore 90
  60      Read L,M,N,O
  70      Data 1,2,3,4
  80      Data 5,6,7,8
  90      Data 9,10,11,12
 100      Print A,B,C,D
 110      Print E,F,G,H
 120      Print R,S,T,U
 130      Print L,M,N,O

>>Run
1          2          3          4
5          6          7          8
1          2          3          4
9         10         11         12

***End***
```

statement: **Data**

format: **Ln DATA A1, A2,...,An**

where:

Ln is a line number.

A1-An are constants, string literals, numeric variables, string variables, numeric expressions, or standard or user defined functions.

The DATA statement specifies values for variables appearing in a READ instruction.

Notes:

1. The READ instruction is used to READ values from DATA statements and assign these values to the READ instruction variable list.

2. The RESTORE instruction allows items in the DATA list to be skipped or reREAD by resetting the DATA list pointer.

Example:

```
10   Data 7*3,9+(2*107),"This is Data",15,75
```

Chapter 14

OUTPUT FORMATTING

This chapter describes the instructions used for formatting output sent to the console or a printer.

instruction: **Print Using**

format: **[Ln] PRINT USING svar, A1,A2,...,An**

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

svar — is a string literal or string variable format definition.

A1-n — are constants, string literals, numeric variables, string variables, numeric expressions, or standard or user defined functions.

The PRINT USING instruction allows the user to specify a format for displaying output.

Note:

In examples in this section the character b represents a blank character.

1. A format field is bounded on either side by any character which is not one of the special characters (# & * + , $! - .). As indicated in the general format for the PRINT USING instruction, a format expression may include more than one format field and may also include string literals. If multiple format fields are specified, the values of expressions are assigned to these fields in order. A format expression may also be assigned to a string variable.

Example:

```
>>List

   10      Num1=11 : Num2=333 : Num3=22
   20      Dim Format$(20) : Format$="###  &&&&M&&&"
   30      Print Using"###  &&&&M&&&",Num1,Num2,Num3
   40      @Using Format$,Num1,Num2,Num3
   50      End

>>Run  11 0333M022  11 0333M022 ***50 End***
```

Statement 30 in the above example outputs the three variables (Num1, Num2, and Num3) with 2 spaces separating the first two and the literal M separating the last two. Any literal which is not one of the PRINT USING special characters may be used to separate format fields. This literal will be PRINTed between the fields if it is a blank or any other character. Statement 40 outputs the same information in the same format using a string variable instead of a string literal in the PRINT USING instruction. Statement 40 also abbreviates PRINT with the symbol @.

2. When a string appears in a PRINT USING list, the characters of the string are PRINTed in the positions held by any of the special format field characters. Strings are left justified in the format field. If the number of characters in the string is less than the number of characters in the format field, the extra spaces will be blank filled. If the number of characters in the string is greater than the number of characters in the format field, the extra characters in the string will be truncated.

Examples:

In the instruction:

```
Print Using "****,**.*", "ABCDEF"
```

the string literal is output in the format:

```
ABCDEFbbb
```

In the instruction:

```
Print Using "&&&,&&&.&&","ABCDEFGHIJKLMN"
```

the string literal is output in the format:

```
ABCDEFGHIJ
```

3. If the number of items in the expression list exceeds the number of specified format fields, the specified format fields will be re-used for the extra items.

Examples:

In the instruction:

```
Print Using "$$&&.&&", A,B,C
```

the expressions A, B, and C will all be PRINTed with the format field $$&&.&&.

In the instruction:

```
Print Using "$$##bb$$&&.&", A,B,C
```

the expressions A and C will be formatted using the format field $$## and the expression B will be formatted using the format field $$&&.&.

4. All normal PRINT functions (such as TAB, SPC, and comma) are overridden by the PRINT USING instruction. The terminating semicolon will still suppress the generation of a new line.

5. The format expression may include a maximum of 128 characters.

6. If a number being formatted has more digits than allowed for in the format expression, an all asterisk error message will result.

Digit Formatting

These special characters may be used to format digits:

#	indicates leading blanks
&	indicates leading zeroes
*	indicates leading asterisks

These symbols are used to right justify digits in a PRINT field. The width of this PRINT field is determined by the number of special characters included in a format field. Any non-digits (such as a minus sign) are eliminated. If the number of special characters in the format field exceeds the number of digits in the expression, the digits will be right justified within the field and preceded

by characters corresponding to the special characters used in the format field. In the following examples, the character b is used to represent blank characters (spaces).

Examples:

Value of expression	Format Field	Output
1	#####	bbbb1
12	#####	bbb12
123	#####	bb123
1234	#####	b1234
12345	#####	12345
123456	#####	*****
1	&&&&&	00001
12	&&&&&	00012
123	&&&&&	00123
1234	&&&&&	01234
12345	&&&&&	12345
123456	&&&&&	*****
1	*****	****1
12	*****	***12
123	*****	**123
1234	*****	*1234
12345	*****	12345
123456	*****	*****

Comma (,)

The comma (,) places a comma in the position in which it appears in the format field. If the format specifies that a comma be output in a position in the field which consists of leading blanks, zeroes, or asterisks, then a blank, a zero, or an asterisk respectively are printed in the comma position.

Examples:

Value of Expression	Format Field	Output
2003	##,###	b2,003
4	##,###	bbbbb4
4457	&&&,&&	044,57
18	&&&,&&	000018
996546	*****,*	99654,6
22	*****,*	****2,2

Decimal Point (.)

The decimal point (.) places a decimal point in the position in which it appears in the format field. All digit positions which follow the decimal point are filled with digits. If the expression contains fewer fractional digits than are specified, zeroes will be PRINTed in the extra positions. If the expression contains more fractional digits than are specified, the expression will be rounded so that the number of fractional digits equals the number of format positions specified.

Examples:

Value of Expression	Format Field	Output
234	###.###	234.000
23.4567	###.###	b23.457
13	&&.&	13.0
66.72319	&&.&	66.7
876.1245	*****.**	**876.12
1234567.245	*****.**	*****.**

In the last example, when too many significant digits appear to the left of a decimal point, an all asterisk error message is generated.

Fixed Plus (+) and Minus (-) Signs

The plus (+) and minus (-) signs may appear in the first character position in a format field. The character + or - will print the respective sign of an expression in the specified character position in the format field. When an expression is preceded by a plus or minus sign, any leading zeroes will be replaced by blanks, zeroes, or asterisks as specified. When a positive expression is preceded by a minus sign, a blank space is left in the sign position.

Examples:

Value of Expression	Format Field	Output
56.8888	+##.###	+56.889
4.564	+##.###	+b4.564
6.456	+&&&.&&	+006.46
234.2	+&&&.&&	+234.20
-23.56	-*****.*	-***23.6
2345	-*****.*	b*2345.0
-2345678.34	-*****.*	******.*

Floating Plus (++) and Minus (--) Signs

The use of two or more plus or minus signs at the beginning of a format field will output the respective sign directly preceding the value of the expression. If a positive expression is PRINTed USING a floating minus format, a blank is PRINTed immediately preceding the number instead of a minus sign. The additional signs in the floating point format can be used to represent digits.

Examples:

Value of Expression	Format Field	Output
2.234	++###.###	b+b2.234
22.234	++###.###	b+22.234
-44.56	--&&&.&	b-044.6
5.32	--&&&.&	bb005.3
178.456	++****.**	b+*178.46
12345678.45	++****.**	******.**
55.17	+++++.++	bb+55.17
55.17	-----.--	bbb55.17
-55.17	-----.--	bb-55.17

Fixed Dollar Sign ($)

The dollar sign ($) is used in either the first or second character position in the format field to PRINT out a dollar sign in that position. A dollar sign specified in the second position of the format field must be preceded by either a plus (+) or a minus (-) sign.

Examples:

Value of Expression	Format Field	Output
23.456	$##.##	$23.46
4.52	$##.##	$b4.52
-57.654	-$&&&.&&&	-$057.654
123.7789	-$&&&.&&&	b$123.779
2.34	$*.*	$2.3
234.55	$*.*	**.*

Floating Dollar Sign ($$)

The use of two or more dollar signs beginning at either the first or second character position in the format field will output a dollar sign immediately preceding the value of an expression. If the dollar signs begin in the second character position, the first character position must contain a plus or a minus sign.

Examples:

Value of Expression	Format Field	Output
234.2345	$$$$#.###	b$234.235
4.45	$$$$#.###	bbb$4.450
23.989	$$&&.&&	b$23.99
4.5	$$&&.&&	b$04.50
24.56	-$$***.**	bb$*24.56
-4455.67	-$$***.**	-$4455.67

Exponent Fields (!!!!)

Four consecutive exclamation characters (!!!!) indicate an exponent in the format field. The exclamation points represent the expression E±nn, where n is any digit. When used with a numeric expression, this format field will output the expression in exponential form.

Examples:

Value of Expression	Format Field	Output
23.3456	##.##!!!!	23.35E+00
2000	##.##!!!!	20.00E+02
-.36	-&&&.&&!!!!	-360.00E-03

7. Only the characters #, &, or ! should be used to the right of a decimal point. The asterisk (*) character follows the same rules as the # character when used to the right of a decimal point.

8. Only one type of floating character may be used within a single format instruction. Either the floating dollar ($$) character, pluses (++), or minuses (--) may be used, but may not be combined.

9. A non-floating PRINT character cannot be placed to the left of a floating character. For example, the format fields $+++ or +$$$ are legal but the format field +$++ is illegal.

10. Structured Basic does not check comma syntax.

11. Only one decimal point may be used in a format field.

12. Trailing + or - signs are illegal.

function: **Space**

format: **SPC(aexp)**

where:

aexp is an arithmetic expression, variable, or constant.

The SPC function instructs Basic to PRINT a specified number of SPaCes.

Notes:

1. The SPC function may be used only in conjunction with a PRINT instruction. If used elsewhere, it is a transparent function:

```
Value = Spc(1.47)
```

is the same as

```
Value = 1.47
```

2. Unlike the TAB function, which always determines PRINT position relative to column 0, the SPC function determines PRINT position relative to the current PRINT column.

Example:

```
>>List

  10          Num1=2
  20          Input Num2,Num3
  30          Print Spc(10);Num1;Spc(Num1*10);Num2;Num3
  40          End

>>Run
? 20,30
              2                    2030

column no.   10                     31
```

In this example, the computer is instructed to skip 10 SPaCes, PRINT Num1, skip 20 SPaCes, and PRINT Num2 and Num3.

function: **Tab**

format: **TAB(aexp)**

where:

aexp is an arithmetic expression, variable, or constant.

The TAB function causes output to begin in a specified column.

Notes:

1. The TAB function may be used only in conjunction with a PRINT instruction. If used elsewhere, it is a transparent function:

```
Value = Tab(6.2)
```

is the same as

```
Value = 6.2
```

2. Multiple TAB functions may be included in one PRINT instruction, but the user should keep in mind that the PRINT position indicated by successive TAB functions is always determined relative to column 0.

3. Columns are numbered 0 through the page width so the first column is column 0.

4. If the argument to the TAB function exceeds the current page width, it is reduced modulo that page width to a number between 0 and the page width. The default value for the page width is 80 characters.

5. If the argument is negative, no TABbing takes place.

6. If the (reduced) argument is greater than the current column position, a new line (RETURN-LINE FEED) is issued and the TAB is executed onto the next line.

Example:

```
>>List

   10          Addtab=1
   20          Input Num1,Num2,Num3
   30          Print Tab(2);Num1;Tab(5);Num2;Tab(Addtab+9);Num3
   40          End

>>Run
? 5,8,9
  5  8     9
***40 End***
```

In this example, the computer is instructed to begin PRINTing Num1 in column 2, Num2 in column 5, and Num3 in column 10.

Chapter 15

INPUT AND OUTPUT TO DISK FILES AND DEVICE DRIVES

Structured Basic treats all input and output as reading and writing data from files rather than specific peripheral devices. Because of this, a single approach can be used to perform all input and output. For example, the programmer goes through the same process to send data to a printer as he does to send it to a disk file.

When the programmer gives an input or output instruction, routines within Basic, called device drivers, handle the differences between the various hardware devices. Those differences are largely invisible to the programmer.

To read or write data, the programmer generally follows these steps:

1. First, a channel to a disk file or a device driver must be opened with the OPEN instruction.

2. Then the programmer uses the INPUT, GET, PRINT, and PUT instructions to read and write data.

3. When the programmer is finished using the device or file, the logical channel is closed with the CLOSE instruction.

The major exception to this scheme is standard input and output to the console. The channel for the console is always open and cannot be closed. Any PRINT or INPUT instruction that doesn't use a channel assigned to a disk file or another device driver automatically uses the standard channel to the console. Chapter 13 describes standard input and output to the console.

Basic also provides a second device driver for the console that is described in this chapter. It differs from the standard console driver in that the full range of console functions and cursor addressing can be used with it. The standard driver doesn't offer these options.

The rest of this chapter describes the use of the input and output instructions.

HOW THE FILES ARE ORGANIZED

Data is information. A file is a place and method for storing many individual items of information.

A computer data file (or file) is defined by:

1. The storage medium (paper tape, floppy disk, etc.),

2. The method of accessing the data (sequential or random),

3. The code by which the data is translated for storage (ASCII or internal machine representation).

Records

A record is a group of related items. A data file is made up of records. Information is inserted into or retrieved from the file record by record.

If a file contained information on all of the baseball games in one year, each record might contain information on one game, one player, or one team. All of the records would contain similar information. If the first record (record number 0) held the statistics on game #1, the first number in the first record could be the number of hits team A got, while the second number would be the number of hits team B got. The third number could be the number of errors made by team A, while the fourth number would be the number of errors made by team B. The second record (record number 1) would contain similar information covering game #2.

Fields

A field describes one item in a record. A field can contain string (alphabetic) or numeric information.

In the above example, instead of saying that the first number was the number of hits team A got, we could say that the first field was numeric and contained the number of hits team A got. We can now further describe the record layout by saying that the fifth and sixth fields are alphabetic and contain the team names.

File Pointer

While a file is OPEN, Basic maintains a pointer which may determine where the next read or write will occur. This pointer can be manipulated by the various file read and write instructions as will be explained in the following paragraphs.

When a file is first OPENed, the File Pointer is positioned just before the first byte of the first record of the file. If the file is a new file (i.e., it contains no data), the beginning of the file and the end of the file coincide, and so the File Pointer also points to the end of file. This is appropriate. If a read is attempted on a new file, an end of file error will be returned because there is no data in the file.

If no record number is specified in a file I/O instruction (sequential I/O), the data list contained in the instruction is written to or read from the file from the current position of the file pointer.

If a record number is specified in a file I/O instruction (random I/O), the File Pointer is moved to a position just before byte zero of the specified record and then the data list contained in the instruction is written to or read from the file from the new position of the File Pointer.

If a byte number is specified in addition to the record number, the File Pointer will be positioned just before the specified byte in the current or specified record before the I/O instruction is executed.

If there is no data list associated with a PUT instruction, the pointer is repositioned as specified above and no input or output takes place.

After the input or output has taken place, the PUT and GET instructions leave the File Pointer just after the byte which was last input or output.

The PRINT and INPUT instructions use the Carriage RETURN or Carriage RETURN-LINE FEED sequence as a delimiter.

The PRINT instruction will move the File Pointer to a position just after the last PRINT character. If the PRINT line starts with byte zero of the current record and is the same length as the specified record size, the pointer will be positioned just before byte zero of the next logical (sequential) record.

The PRINT instruction will automatically insert a RETURN-LINE FEED sequence if:

1. The data list associated with the Print instruction will cause a record to be output which is longer than the page width (refer to the SET instruction), and

2. No single item in the data list is longer than the specified page width.

This can happen when items in a PRINT list are separated by commas or a PRINT instruction is terminated with a comma or semicolon and a subsequent PRINT instruction adds to the already PRINTed line. This rule also applies to the PRINT instruction when used to send output to the terminal.

A PRINT list containing an item which is longer than the page width will cause a run time error. The default page width is 80 characters. It is possible to change this parameter by using the SET instruction.

Sequential Files

A sequential file is written in the order of the record numbers. That is, record number zero is written with the first output (PUT or PRINT) instruction, record number one is written next, then record number two, etc. This continues until the file is CLOSEd. When the file is OPENed again, the first record which is output will write over record number zero, etc.

When a file is read sequentially, record zero is read first, then record one, etc.

Note that if an entire record is not input or output each time the file is accessed, the File Pointer will remain after the last character which was read or written. Any subsequent file access will read or write from that position, not from the beginning of the next record.

Random Files

A random access file is written in the order specified by the programmer. Each output instruction must specify the number of the record which is to be written. When a file is read randomly, the programmer must specify the record number to be read for each INPUT or GET instruction.

A file which was written sequentially may be read as a random file. Files which were written randomly may be read by sequential or random instructions provided that no attempt is made to read a record before it has been written.

A file may be accessed by any combination of sequential and random instructions.

A random instruction, one that specifies a record or record and byte number, will reposition the File Pointer before accessing the file.

A sequential instruction, one that does not specify a record number, will access the file from the current position of the File Pointer.

Internal Machine Vs. ASCII Representation

Basic can store data in two formats. Both consist of groups of ones and zeros (binary representation), but each needs to be translated differently for output to a terminal or printer.

ASCII (American Standard Code for Information Interchange) is the closest to written letters and numbers. ASCII is stored in the machine according to the table of ASCII codes (see Appendix B). Each character code occupies one byte (8 bits). When the code is translated from binary to decimal, the character which is represented can be found in the ASCII table. This is the way all Basic strings are stored. ASCII information is stored so that one character is stored in one byte.

The need arises for another method of storing numbers when speed of arithmetic computation, Input/Output translation, and amount of storage space become important factors.

Internal Machine code is used only for the storage of numbers. The number of bytes occupied by a number depends on its magnitude and precision (the size of the number and the number of decimal places which need to be kept.) There are three internal machine formats:

Integer	size:	2 bytes
	range:	+32767 < N < -32768
	accuracy:	nearest integer
Short Floating Point	size:	4 bytes
	range:	±9.99E+62 < N < ±9.99E-65
	accuracy:	6 decimal digits
Long Floating Point	size:	8 bytes
	range:	±9.99E+62 < N < ±9.99E-65
	accuracy:	14 decimal digits

Refer to Chapter 3 for a more complete discussion of internal machine representation.

DIFFERENCES IN THE INPUT AND OUTPUT INSTRUCTIONS

The following paragraphs describe the major differences between the two sets of input and output instructions: PRINT-INPUT and PUT-GET. In addition to the differences described here, the two sets of instructions also use the file pointer somewhat differently. See the description of the file pointer earlier in this chapter for more information.

PRINT and INPUT

PRINT and INPUT write and read in ASCII format. These instructions are primarily intended to write data to an output file which is to be displayed on a console or printer, and to read a data file which was created by the Screen Editor. When used with a file, they exactly parallel their use with the console terminal. (See Chapter 13 for information on their use with the console.) In particular, INPUT requires a comma or RETURN between INPUT data items.

The PRINT and INPUT instructions translate numeric data from ASCII to Internal Machine Format (or vice versa). They are therefore slower than the PUT and GET instructions. There is little need for the programmer to keep track of the type of variable which is written out with a PRINT instruction. An integer variable which is written using the PRINT instruction may be INPUT back into a string or floating point variable without losing its meaning. No attempt should be made to INPUT an ASCII string into a numeric variable. A file which has been output using the PRINT instruction may also be Typed out (using the operating system) and read whereas the same file output with the PUT instruction would not be able to be Typed.

PUT and GET

It is recommended that the PUT and GET instructions be used with disk data files that are created and used by Basic programs.

The PUT and GET instructions write and read in internal machine format. If an integer variable is written out to a file using a PUT instruction, it must be read back into an integer variable (and not into a floating point variable or an ASCII string) using a GET instruction or it will be meaningless. The machine does no translation with these instructions so they can be executed faster than PRINT and INPUT. The programmer must keep track of the types and lengths of variables written by PUT so that they may be read back in properly.

Input and Output with Character Strings

A string (which is stored internally in ASCII code) may be output with a PUT or PRINT instruction with the same general result (except that PRINT outputs a carriage RETURN-LINE FEED sequence where appropriate and tabs, using spaces, when variables are separated by commas). This is because PUT will output a string in internal machine format, which (for a string) is ASCII. PRINT will not translate the string because it is already in ASCII format.

USING THE DEVICE DRIVERS

Structured Basic offers four standard device drivers. They are:

Standard Console Driver

The standard console driver sends input and output directly to the console without going through the operating system. The channel to this driver is always open and cannot be closed. To use this driver with the PRINT and INPUT instructions, do not specify a channel number. To use this driver with the PUT and GET instructions, specify channel number 0. These instructions would send output to the console and then read in data:

```
Print"The current balance is: ";Balance
Input"Do you wish to continue?",Answer$
```

$CO Console Driver

The $CO console driver sends input and output to the console through the operating system. This allows the input and output instructions to use the console system calls to invoke special console functions and to use cursor addressing. (See the GET, PUT, PRINT, and INPUT instructions for more on these features.) To use the $CO driver, use the OPEN instruction to open a file named $CO. These instructions show the use of the $CO driver:

```
 5 Rem:  Line 10 opens a channel to driver $CO
10 Open\1\"$CO"
15 Rem:  Line 20 has $CO clear the screen
20 Print\1,0,0\
25 Rem:  Line 30 has $CO begin printing at column 30, row 12
30 Print\1,30,12\"Enter Password -->"
35 Rem:  Line 40 has $CO begin a protected field
40 Print\1,15,0\
45 Rem:  Line 50 has $CO read the password beginning at
46 Rem:  column 48, row 12
50 Input\1,48,12\Password$
55 Rem:  Line 60 has $CO end the protected field
60 Print\1,16,0\
65 Rem:  Line 70 has $CO clear the screen
70 Print\1,0,0\
75 Rem:  Line 80 closes the channel to $CO
80 Close\1\
```

Disk Drivers

The disk device drivers send data to and from files on disks. To use these drivers, use the OPEN instruction to open each disk file to be used. Then these drivers are automatically used each time the input or output is sent to the files. The following instructions show the use of the disk drivers:

```
1 Temporary'value=1.0
5 Rem:  Line 10 creates the file "scratch"
10 Create"scratch"
15 Rem:  Line 20 opens a channel to file "scratch"
20 Open\2\"scratch"
25 Rem:  Line 30 PUTs a value into the file
30 Put\2\Temporary'value
35 Rem:  Line 40 GETs record 0 from the file
40 Get\2,0\Read'value
50 Print Read'value
55 Rem:  Line 60 closes the channel
60 Close\2\
65 Rem:  Line 70 deletes the file "scratch" from the disk
70 Erase"scratch"
```

The following Basic instructions are useful in working with disk files:

Create Creates a disk file
Ren and
Rename Rename a disk file
Erase Deletes a disk file

Before a disk file can be used (OPENed), it must already exist on the disk.

Line Printer Driver

The line printer driver sends output to the system line printer. To use this driver, use the OPEN instruction to open a file named $LP. The following instructions show the use of the $LP driver:

```
10 Open\5\"$LP"
20 Print\5\Buyer$,Price,Profit
30 Close\5\
```

USING CHANNELS

The OPEN instruction reserves an Input/Output channel and assigns the I/O channel number to the file reference or file device. Once a file has been OPENed (and assigned a channel number) all input and output from/to that file will be done through the assigned channel.

Basic normally carries 8 I/O channels in addition to the console and the Procedure Library. One I/O channel is needed for each file which is OPENed at the same time. Each channel occupies 192 bytes of memory. Increasing the number of channels allows more files to be OPENed simultaneously but reduces the amount of memory available to the Basic user. Conversely, decreasing the number of I/O channels increases the amount of available memory. If you wish to change the number of channels available for use, see the Appendix, Changing the Number of I/O Channels.

instruction: **Create File**

format: **[Ln] CREATE file-ref**

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

file-ref — is a Cromix path name composed of optional directory names and a file name separated by slashes. This may be a string literal enclosed in quotation marks or a string variable.

The CREATE instruction places the file name (and file name extension, if one is used) in the specified directory. A file may only be CREATEd once, and must be CREATEd before it can be OPENed.

Notes:

1. The file path name is composed of optional directory names and/or the file name and/or the file name extension. These may be a string variable or a string literal enclosed in quotation marks.

2. The file name and file name extension may include any printable ASCII character except the following:

   $ * ? = . , : - "space"

3. No space is allocated to a file by the CREATE instruction. All file space is dynamically allocated only when needed.

4. Error Number 137 (File Already Exists) will result if the file already exists in the directory when the CREATE instruction is given.

instruction: **Open File**

format: **[Ln] OPEN\n\ file-ref**

[Ln] OPEN\n,p1\ file-ref

[Ln] OPEN\n,p1,p2\ file-ref

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n — is the required file (channel) number.

p1 — (for a disk file) is the optional record size in bytes which may be assigned any value between 1 and 32,767. The default value is 128 bytes per record (one sector).

p2 — (for a disk file) is the optional file access mode specifier:

=1 is read only
=2 is write only
=3 is read and write
(default value)

file-ref — is a Cromix path name composed of optional directory names and a file name separated by slashes. This is the name of the device driver or disk file pathname. This may be a string literal enclosed in quotation marks or a string variable.

The OPEN instruction allows a disk file or system device to be linked to a file number (channel) for future reference in connection with file Input/Output instructions (i.e., PUT, GET, INPUT, PRINT, CLOSE).

Notes:

1. The file path name is composed of optional directory names and/or the file name and/or the file name extension. These may be a string variable or a string literal enclosed in quotation marks.

2. The file name and file name extension may include any printable ASCII character except the following:

$ * ? = . , : - "space"

3. The file number must be between 1 and the maximum channel number available with an absolute maximum of 16. As Basic is shipped, the maximum channel number is 8. See Appendix F for the method of changing the number of channels available.

4. Error number 133 (File Number) will result if an attempt is made to reference a file number greater than the maximum channel number available.

5. The file number 0 (zero) is reserved for the console. It cannot be OPENed by the user. All input/output (GET, INPUT, PUT, and PRINT) directed from/to file 0 will use the console.

   Caution: If file number 0 is specified by the CLOSE instruction, all currently OPENed files will be CLOSEd.

6. Although files OPENed for read/write access may be used as write only files, slightly faster execution speed may result if the file is OPENed for write only access.

7. If the PRINT instruction is used to write a single item which is longer than the current page width, it will be necessary to increase the page width (using the SET instruction) in order to avoid Error number 6 (Print Item Size). Refer to the discussion section of this chapter for additional information.

instruction: **Close File**

format: **[Ln] CLOSE**

[Ln] CLOSE \n

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n — is an optional file (channel) number. The default value if unspecified is all currently OPENed files.

The CLOSE instruction disassociates the channel number and file which were associated by the OPEN instruction.

Note:

1. If the CLOSE instruction is given without a file number or with the file number set equal to zero, all currently OPENed files will be CLOSEd.

instruction: **Erase File**

format: **[Ln] ERASE file-ref**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

file-ref is a Cromix path name composed of optional directory names and a file name separated by slashes. This may be a string literal enclosed in quotation marks or a string variable.

The ERASE instruction will remove a disk file from the file directory.

Examples:

```
Erase "TEMP.BAS"
Erase "*.BAK"
Erase "/USR/TEMP.REL"
```

The first of these examples will ERASE the file named TEMP.BAS from the current or default disk drive. The second will ERASE all files from the current disk drive with the file name extension of BAK. The third example will ERASE the file called TEMP.REL from the /usr directory.

instruction: **Rename File**

format: **[Ln] RENAME old-file-ref, new-file-ref**

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

file-ref-1 — is a Cromix path name composed of optional directory names and a file name separated by slashes. This may be a string literal enclosed in quotation marks or a string variable.

file-ref-2 — is a Cromix path name composed of optional directory names and a file name separated by slashes. This may be a string literal enclosed in quotation marks or a string variable.

The RENAME instruction is used to give a new file name to an existing disk file.

Note:

1. The REN instruction performs the same function as the RENAME instruction. The REN instruction is included to be compatible with the CDOS REN command. The parameter order on the REN instruction is the opposite of the RENAME instruction. RENAME is the preferred form for the Cromix Operating System.

Example:

```
Rename "Oldfile", "Newfile"
```

This command will change the name of the file (on the current disk drive) from OLDFILE to NEWFILE.

instruction: **Rename File**

format: **[Ln] REN new-file-ref, old-file-ref**

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

new-file-ref — is the new name for the file. This is a Cromix path name composed of optional directory names and a file name separated by slashes. The file name may be a string literal enclosed in quotation marks or a string variable name.

old-file-ref — is the name of an existing file. This is a Cromix path name composed of optional directory names and a file name separated by slashes. The file name may be a string literal enclosed in quotation marks or a string variable name.

The REN instruction is used to give a new file name to an existing file on the directory.

Note:

1. The REN instruction performs the same function as the RENAME instruction. The REN instruction is included to be compatible with the CDOS REN command. The parameter order on the REN instruction is the opposite of the RENAME instruction.

Example:

```
Ren "Newfile", "Oldfile"
```

This command will change the name of the file (on the current disk drive) from NEWFILE to OLDFILE.

instruction: **Print**

format: **[Ln] PRINT**

[Ln] PRINT exp-1,...,exp-n

[Ln] PRINT\n

[Ln] PRINT\n,p1

[Ln] PRINT\n,p1,p2

[Ln] PRINT\n\ exp-1,...,exp-n

[Ln] PRINT\n,p1\ exp,...,exp-n

[Ln] PRINT\n,p1,p2\ exp-1,...,exp-n

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n — is an optional file number. If no file number is specified, or if the file number is zero, output goes to the console terminal.

p1 — (for a disk file) is the optional record number. The default value is sequential access starting with record 0 or the current position of the File Pointer.

(for a console) is an optional console function parameter. If specified, this parameter can specify either a special terminal function or a cursor location on the terminal. To be used, the $CO console driver must be the driver used for this instruction. If this parameter isn't specified, the instruction executes without changing the cursor location or invoking any special functions.

p2 — (for a disk file) is the optional byte number. The default value is byte 0.

(for a console) is an optional console function parameter. If specified, this parameter can specify either a special terminal function or a cursor location on the terminal. To be used, the $CO console driver must be the driver used for this instruction. If this parameter isn't specified but p1 is specified, this parameter has a default value of zero. If neither this parameter nor p1 is specified, the instruction executes without changing the cursor location or invoking any special functions.

exp1-n is an optional list of expressions, numeric or string variables, or string literals. The data list must be separated by either semicolons or commas. If an item is followed by a comma, the item following it will be printed starting in the next tab position; if it is followed by a semicolon no space will be left before printing the next item.

When used without a file reference, the PRINT instruction will cause the data list to be output to the console. When used with a file reference, it will cause the data list to be output to an ASCII device (e.g., the line printer) or a disk file in ASCII format.

Notes:

It is recommended that PRINT only be used for the output of PRINT or list files and not be used for writing data files which are to be read back by a Basic program.

1. The at (@) sign may be used to abbreviate the word PRINT in this instruction.

2. Most of the above forms of the PRINT instruction may incorporate the PRINT USING feature. Refer to PRINT USING for additional information.

3. The PRINT instruction outputs a RETURN-LINE FEED (ASCII 13,10) sequence at the end of the data list. If the file is to be read in using the INPUT instruction, a RETURN-LINE FEED sequence (or just a RETURN) must follow each item which is output, as it is this character or character sequence which delimits (terminates) each item. In Basic, there are two ways this can be accomplished:

   a. Allow only one item per PRINT instruction.

   b. Insert a RETURN between each item as follows:

```
100   Cr$ = CHR$(13)
200   Print \1\ "String"; Cr$; Num; Cr$; Text$
```

   Statement 100 assigns the ASCII value of the RETURN to the string variable Cr$. Statement 200 PRINTs a string literal, a numeric variable, and a string variable, each separated by the RETURN.

   The above procedure will allow all of the items output by the PRINT instruction to be read by the INPUT instruction which looks for a RETURN between each item read.

If only numeric data is to be PRINTed and INPUT a comma (ASCII 44) may be used as a delimiter between variables within one INPUT list. The last item in the data list must be delimited by a RETURN.

4. Specifying a negative number for either p1 or p2 will result in the default value being assigned to that parameter.

5. The PRINT instruction, when given without a data list, causes one blank line to be sent to the console or PRINT file.

6. When using the $CO console driver, the p1 and p2 parameters can be used as shown in the following table:

Function	p1	p2
Address cursor on screen	1-80	1-24
Clear screen	0	0
Home cursor without clearing	1	0
Cursor left one character position	2	0
Cursor right one character position	3	0
Cursor up one position	4	0
Cursor down one line	5	0
Clear from cursor to end of line	6	0
Clear from cursor to end of screen	7	0
Set intensity to high light	8	0
Set intensity to low light	9	0
Set intensity to normal light	10	0
Enable keyboard	11	0
Disable keyboard	12	0
Dynamic function keys	13	0
Static function keys	14	0
Begin protected field	15	0
End protected field	16	0
Begin blinking characters	17	0
End blinking characters	18	0
Send from cursor to end of line	19	0
Send from cursor to end of screen	20	0
Transmit screen out auxiliary port	21	0
Delete character at current cursor position	22	0
Insert character at current cursor position	23	0
Delete line at current cursor position	24	0
Insert line at present cursor position	25	0
Formatted screen on	26	0
Formatted screen off	27	0
Begin reverse background field	28	0
End reverse background field	29	0
Begin underlining characters	30	0
End underlining characters	31	0
Display message on	32	0
Display message off	33	0
Insert character off	35	0
Graphics mode on	36	0
Graphics mode off	37	0
Cursor on	38	0
Cursor off	39	0
Memory lock on	40	0
Memory lock off	41	0
Alarm on	45	0
Alarm off	46	0

There are no functions that have a p1 value of 34, 42, 43, or 44.

instruction: **Input**

format: **[Ln] INPUT var-1,...,var-n**

[Ln] INPUT\n\ var-1,...,var-n

[Ln] INPUT\n,p1\ var,...,var-n

[Ln] INPUT\n,p1,p2\ var-1,...,var-n

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n — is an optional file number. If no file number is specified, or if the file number is zero, input is received from the console terminal.

p1 — (for a disk file) is the optional record number. The default value is sequential access starting with record 0 or the current position of the File Pointer.

(for a console) is an optional console function parameter. If specified, this parameter can specify either a special terminal function or a cursor location on the terminal. To be used, the $CO console driver must be the driver used for this instruction. If this parameter isn't specified, the instruction executes without changing the cursor location or invoking any special functions.

p2 — (for a disk file) is the optional byte number. The default value is byte 0.

(for a console) is an optional console function parameter. If specified, this parameter can specify either a special terminal function or a cursor location on the terminal. To be used, the $CO console driver must be the driver used for this instruction. If this parameter isn't specified but p1 is specified, this parameter has a default value of zero. If neither this parameter nor p1 is specified, the instruction executes without changing the cursor location or invoking any special functions.

var1-n — is a list of one or more numeric and/or string variables.

When used without a file reference, the INPUT instruction will cause the data list to be INPUT from the console. When used with a file reference, it will cause the list to be INPUT from an ASCII device (e.g., paper tape reader) or a disk file in ASCII format.

Notes:

1. When a string variable is used with an INPUT instruction, the portion of the string which is referenced is set equal to null characters before the source is moved into the string variable. The whole string is referenced if no subscripts follow the string variable, while various substrings may be referenced by the use of subscripts. Refer to the section **Referencing String Variables** in Chapter 5, for a complete discussion of the subject. The INPUT instruction will not move more characters than can be accepted by the destination string or substring which is being referenced.

2. INPUT can accept no more than 132 characters per line.

3. INPUT retrieves only 7 bit ASCII. The GET instruction must be used to retrieve all 8 bits.

4. INPUT treats some control characters as editing or end of file commands.

5. When using the $CO console driver, the p1 and p2 parameters can be used as shown in the following table:

Function	p1	p2
Address cursor on screen	1-80	1-24
Clear screen	0	0
Home cursor without clearing	1	0
Cursor left one character position	2	0
Cursor right one character position	3	0
Cursor up one position	4	0
Cursor down one line	5	0
Clear from cursor to end of line	6	0
Clear from cursor to end of screen	7	0
Set intensity to high light	8	0
Set intensity to low light	9	0
Set intensity to normal light	10	0
Enable keyboard	11	0
Disable keyboard	12	0
Dynamic function keys	13	0
Static function keys	14	0
Begin protected field	15	0
End protected field	16	0
Begin blinking characters	17	0
End blinking characters	18	0
Send from cursor to end of line	19	0
Send from cursor to end of screen	20	0
Transmit screen out auxiliary port	21	0
Delete character at current cursor position	22	0
Insert character at current cursor position	23	0
Delete line at current cursor position	24	0
Insert line at present cursor position	25	0
Formatted screen on	26	0
Formatted screen off	27	0
Begin reverse background field	28	0
End reverse background field	29	0
Begin underlining characters	30	0
End underlining characters	31	0
Display message on	32	0
Display message off	33	0
Insert character off	35	0
Graphics mode on	36	0
Graphics mode off	37	0
Cursor on	38	0
Cursor off	39	0
Memory lock on	40	0
Memory lock off	41	0
Alarm on	45	0
Alarm off	46	0

There are no functions that have a p1 value of 34, 42, 43, or 44.

instruction: **Put Record**

format: **[Ln] PUT\n**

[Ln] PUT\n\ exp-1,...,exp-n

[Ln] PUT\n,p1\ exp,...,exp-n

[Ln] PUT\n,p1,p2\ exp-1,...,exp-n

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n — is a file number. If file number 0 is used, output will be sent to the console.

p1 — (for a disk file) is the optional record number. The default value is sequential access starting with record 0 or the current position of the File Pointer.

(for a console) is an optional console function parameter. If specified, this parameter can specify either a special terminal function or a cursor location on the terminal. To be used, the $CO console driver must be the driver used for this instruction. If this parameter isn't specified, the instruction executes without changing the cursor location or invoking any special functions.

p2 — (for a disk file) is the optional byte number. The default value is byte 0.

(for a console) is an optional console function parameter. If specified, this parameter can specify either a special terminal function or a cursor location on the terminal. To be used, the $CO console driver must be the driver used for this instruction. If this parameter isn't specified but p1 is specified, this parameter has a default value of zero. If neither this parameter nor p1 is specified, the instruction executes without changing the cursor location or invoking any special functions.

exp1-n — is a list of one or more numeric and/or string variables.

The PUT instruction outPUTs the data in the data list (exp-1,...exp-n) to the file specified by the file number (n). The data is output in internal machine format. This is useful if the data is to be read back by Basic. The PRINT instruction should be used if the file is to be Typed or LISTed to the printer.

Notes:

1. If no outPUT list is included, only the device status is set (i.e., record and byte position on a disk file). This is a useful way of setting status (position) without actually initiating any data transfer.

2. The PUT instruction will output data to a file in internal machine format (refer to the discussion at the beginning of this chapter on Internal Machine vs. ASCII Representation). Numeric data which has been output using the PUT instruction must be read back in using the GET instruction.

3. ASCII data which is output by the PUT instruction will not be readable by INPUT unless a RETURN appears at least every 132 bytes.

4. When using the $CO console driver, the p1 and p2 parameters can be used as shown in the following table:

Function	p1	p2
Address cursor on screen	1-80	1-24
Clear screen	0	0
Home cursor without clearing	1	0
Cursor left one character position	2	0
Cursor right one character position	3	0
Cursor up one position	4	0
Cursor down one line	5	0
Clear from cursor to end of line	6	0
Clear from cursor to end of screen	7	0
Set intensity to high light	8	0
Set intensity to low light	9	0
Set intensity to normal light	10	0
Enable keyboard	11	0
Disable keyboard	12	0
Dynamic function keys	13	0
Static function keys	14	0
Begin protected field	15	0
End protected field	16	0
Begin blinking characters	17	0
End blinking characters	18	0
Send from cursor to end of line	19	0
Send from cursor to end of screen	20	0
Transmit screen out auxiliary port	21	0
Delete character at current cursor position	22	0
Insert character at current cursor position	23	0
Delete line at current cursor position	24	0
Insert line at present cursor position	25	0
Formatted screen on	26	0
Formatted screen off	27	0
Begin reverse background field	28	0
End reverse background field	29	0
Begin underlining characters	30	0
End underlining characters	31	0
Display message on	32	0
Display message off	33	0
Insert character off	35	0
Graphics mode on	36	0
Graphics mode off	37	0
Cursor on	38	0
Cursor off	39	0
Memory lock on	40	0
Memory lock off	41	0
Alarm on	45	0
Alarm off	46	0

There are no functions that have a p1 value of 34, 42, 43, or 44.

instruction: **Get Record**

format: **[Ln] GET\n\ exp-1,...,exp-n**

[Ln] GET\n,p1\ exp-1,...,exp-n

[Ln] GET\n,p1,p2\ exp-1,...,exp-n

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n — is a file number. If file number 0 is used, input will be accepted from the console.

p1 — (for a disk file) is the optional record number. The default value is sequential access starting with record 0.

(for a console) is an optional console function parameter. If specified, this parameter can specify either a special terminal function or a cursor location on the terminal. To be used, the $CO console driver must be the driver used for this instruction. If this parameter isn't specified, the instruction executes without changing the cursor location or invoking any special functions.

p2 — (for a disk file) is the optional byte number. The default value is byte 0.

(for a console) is an optional console function parameter. If specified, this parameter can specify either a special terminal function or a cursor location on the terminal. To be used, the $CO console driver must be the driver used for this instruction. If this parameter isn't specified but p1 is specified, this parameter has a default value of zero. If neither this parameter nor p1 is specified, the instruction executes without changing the cursor location or invoking any special functions.

exp1-n — is a list of one or more numeric and/or string variables.

The GET instruction GETs the data in the data list (exp-1,...exp-n) from the file specified by the file number (n). The data is input without translation (internal machine format). This is useful if the data has been output by the PUT instruction.

Notes:

1. When a string variable is used with a GET instruction, the portion of the string which is referenced is set equal to null characters before the source is moved into the string variable. The whole string is referenced if no subscripts follow the string variable, while various substrings may be referenced by the use of subscripts. Refer to the section **Referencing String Variables** in Chapter 5, for a complete discussion of the subject. The GET instruction will not move more characters than can be accepted by the destination string or substring which is being referenced.

2. When using the $CO console driver, the p1 and p2 parameters can be used as shown in the following table:

Function	p1	p2
Address cursor on screen	1-80	1-24
Clear screen	0	0
Home cursor without clearing	1	0
Cursor left one character position	2	0
Cursor right one character position	3	0
Cursor up one position	4	0
Cursor down one line	5	0
Clear from cursor to end of line	6	0
Clear from cursor to end of screen	7	0
Set intensity to high light	8	0
Set intensity to low light	9	0
Set intensity to normal light	10	0
Enable keyboard	11	0
Disable keyboard	12	0
Dynamic function keys	13	0
Static function keys	14	0
Begin protected field	15	0
End protected field	16	0
Begin blinking characters	17	0
End blinking characters	18	0
Send from cursor to end of line	19	0
Send from cursor to end of screen	20	0
Transmit screen out auxiliary port	21	0
Delete character at current cursor position	22	0
Insert character at current cursor position	23	0
Delete line at current cursor position	24	0
Insert line at present cursor position	25	0
Formatted screen on	26	0
Formatted screen off	27	0
Begin reverse background field	28	0
End reverse background field	29	0
Begin underlining characters	30	0
End underlining characters	31	0
Display message on	32	0
Display message off	33	0
Insert character off	35	0
Graphics mode on	36	0
Graphics mode off	37	0
Cursor on	38	0
Cursor off	39	0
Memory lock on	40	0
Memory lock off	41	0
Alarm on	45	0
Alarm off	46	0

There are no functions that have a p1 value of 34, 42, 43, or 44.

Example Program - Random Access Files

```
 10  Rem This is a program which demonstrates
 20  Rem the implementation of a Random
 30  Rem Access File.
 40  Rem
 50  Rem The program creates and opens a file, writes
 60  Rem 51 records sequentially, allows
 70  Rem the user to repeatedly select any
 80  Rem of the 51 records at random, and
 90  Rem then closes and erases the file.
100  Rem
110  Dim Text$(21)
120  Integer Index,Number
130  Create"Randtest"
140  Open1,24"Randtest"
150  On Error Goto Query
160  On Esc Goto Finish
170  Print"Pause, writing file Randtest"
180    For Index=0 To 50
190    Put1"This is record number ",Index
200    Next Index
210 *Query :  Print
220  Print"Which record would you like to see?"
230  Print"Enter record number (0-50) or -1 to stop:  ";
240  Input Index
250  If Index<0 Then Goto Finish
260  If Index>50 Then Goto Query
270  Get1,IndexText$(-1),Number
280  Print"Contents of record number ";Index;" is:"
290  Print Text$;Number
300  Print
310  Goto Query
320 *Finish :  Close
330  Erase"Randtest"
340  Print :  Print :  Print"File Randtest erased"
350  End
```

On line 130, the file Randtest is CREATEd. If a file does not exist in the directory, it must be CREATEd before it can be OPENed for reading or writing. The file is OPENed on line 140 with a record length of 24 bytes which are allocated as follows:

22 bytes for the string, and
2 bytes for an integer

Line 150 ensures that any run time error (such as an invalid user response) will return control to the user within the program and will not cause the program to terminate abnormally. Because many programs can be terminated by the use of the ESCAPE key, statement 160 is included in this program. The program will

still be terminated by depressing the ESCAPE key, but the active file will be closed and deleted before control is returned to the user.

Statements 180 through 200 write out 51 records, each containing (for identification) the record number in addition to a string. Because no record numbers are specified, the records are written sequentially starting with record number zero.

The user is then asked for the number of the record to be retrieved or displayed and on line 270 the record whose number is specified by the variable I is retrieved. Because the records which were written to this file each contained a 22 character string followed by an integer number, they must be read back into variables of the same type and length.

The user is allowed to view as many records as desired. Entering a negative one (-1) when asked for a record number will cause the file to be CLOSEd and ERASEd and program execution to be terminated.

Refer to the discussion section of this chapter for more information on the use of files.

Chapter 16

FUNCTIONS

Cromemco Structured Basic includes a number of functions, such as arithmetic and trigonometric functions, that perform common, frequently used calculations. These functions are pre-defined in Basic so that the programmer does not need to write a program every time one of these functions is required.

Cromemco Basic also includes a number of functions designed to increase string handling capabilities, system functions which provide general system information, and functions which makes it possible to call assembly language subroutines.

In addition to these pre-defined functions, Cromemco Basic permits the programmer to define additional functions.

This chapter describes how to write user defined functions, and how to use the arithmetic, trigonometric, and string functions. Chapter 17 describes several functions used to determine system and file status, and Chapter 18 describes several functions used for machine level instructions.

WRITING PROGRAMMER DEFINED FUNCTIONS

The DEF FNS instruction allows users to write their own functions.

function: **Programmer Defined Function**

format: **DEF FNS(avar-1,avar-2,...,avar-n)=aexp**

where:

S is any legal arithmetic variable name.

avar1-n are arithmetic variables.

aexp is an arithmetic expression, variable, or constant.

The DEF FuNction permits the programmer to define functions in addition to the pre-defined functions included in Basic.

Notes:

1. The definition of a function must be a statement which is encountered during the execution of a program in order for Basic to retain the definition. After the function has been defined, it may be used in a command line.

2. Any of the variables used in the definition of a function (avar-1 through avar-n) are unique to the function. Their use in the definition does not conflict with a variable of the same name appearing elsewhere in the program.

   If a variable appears on the right side of the definition (as a part of aexp) without appearing on the left, its value may be accessed and changed outside the function definition and its value is maintained throughout a function call. These variables are not unique to the function.

Examples:

```
>>List

   10      Length=20
   20      Width=10
   30      Height=5
   40      Def Fnvolume(A,B,C)=A*B*C
   50      Print Fnvolume(Length,Width,Height)
   60      End

>>Run
1000
***60 End***
```

In the above example, variables A, B, and C are unique to the definition of the function. Variables Height, Length, and Width are regular program variables whose values may be established at any point in the program (Partition).

The following example should help to clear up any confusion about variables which are unique to a function definition.

```
>>List

  100     Def Fnmult(P'meter) = P'meter * Factor
  200     P'meter = 77
  300     Factor = 5
  400     Actual'value = 10
  500     Print Fnmult(Actual'value)
  600     Print P'meter
  700     End

>>Run
50
77
***700 End***
```

In the above example, statement 100 defines the function Fnmult in terms of the formal parameter (function definition variable) P'meter and the program variable Factor. In statement 100 P'meter is called a function definition variable because it appears to the left of the equal sign in the function definition. As such, P'meter can not be accessed by the user. The program variable (P'meter) defined by statement 200 is a different variable than the aforementioned function definition variable, and can be accessed by the user. According to the above definitions, Factor, appearing in statement 100, is a program variable. In statements 300, 400, and 500 program variables P'meter, Factor, and Actual'value are assigned values.

In statement 500, the value of the function Fnmult is computed using the value of Actual'value to replace P'meter in the definition of the function. Being a program variable, Factor maintains its value in the calculation of the value of the function. Once the value of the function has been calculated, its value is printed.

Statement 600 prints the value of the program variable P'meter to demonstrate that the program variable P'meter can be accessed by the user, that it maintains its value through the function call, and that it is a different variable than the function definition variable P'meter.

ARITHMETIC FUNCTIONS

Structured Basic offers these arithmetic functions:

Arithmetic Functions

Abs(X)	absolute value of X
Binadd(X,Y)	binary addition
Binand(X,Y)	binary logical And
Binor(X,Y)	binary logical Or
Binsub(X,Y)	binary subtraction
Binxor(X,Y)	binary logical Exclusive Or
Exp(X)	"e" to the power X
Fra(X)	fractional portion of X
Int(X)	integer value of X
Irn(X)	generates an integer random number between 0 and 32767
Log(X)	natural logarithm of X
Max(X1,.,Xn)	returns the numeric expression Xn with the maximum value in the expression list
Min(X1,.,Xn)	returns the numeric expression Xn with the minimum value in the expression list
Randomize	used with Rnd and Irn to produce different sets of random numbers
Rnd(X)	generates a random number between 0 and 1
Sgn(X)	algebraic sign of X
Sqr(X)	square root of X

function: **Absolute Value**

format: **ABS(aexp)**

where:

aexp is an arithmetic expression, variable, or constant.

The ABS function gives the ABSolute (i.e., positive) value of aexp, which can be any arithmetic expression.

Example:

```
>>List

   10      Print Abs(-26), Abs(26)
   20      End

>>Run
26                    26
***20 End***
```

functions: **Binary Operations**

format: **BINADD(aexp1,aexp2)**

BINAND(aexp1,aexp2)

BINOR(aexp1,aexp2)

BINSUB(aexp1,aexp2)

BINXOR(aexp1,aexp2)

where:

aexp1-1 and aexp1-2 are arithmetic expressions, variables, or constants.

The BINAND, BINOR, BINXOR functions perform logical operations bit by bit on 16-bit operands. The BINADD and BINSUB functions perform binary arithmetic operations on integer (16-bit) operands.

BINADD performs the BINary ADDition of aexp1 and aexp2. A carry is ignored.

BINAND performs a BINary AND logical operation on aexp1 and aexp2. It returns a 1 bit in a given position if both bits are equal to 1 and a 0 otherwise.

BINOR performs a BINary OR logical operation on aexp1 and aexp2. It returns a 1 bit in a given position if either bit is equal to 1 and a 0 otherwise.

BINSUB performs the BINary SUBtraction of aexp1 minus aexp2. An overflow is ignored.

BINXOR performs a BINary EXclusive OR logical operation on aexp1 and aexp2. It returns a 1 bit in a given position if either bit is equal to 1 and the other bit is equal to 0. A 0 is returned otherwise.

Notes:

1. If necessary, aexp1 and/or aexp2 will be converted to 16 bit integers for the purpose of these functions.

2. Refer to Chapter 6 for a discussion of Boolean operators.

function: **Exponent**

format: **EXP(aexp)**

where:

aexp is an arithmetic expression, variable or constant.

The EXP function calculates the value of the constant e (where e = 2.71828...) raised to the aexpth power.

Example:

```
>>List

   10      Num1=4.1
   20      Num2=Exp(Num1)
   30      Print Num2
   40      End

>>Run
60.340287597344
***40 End***
```

function: **Fractional Portion**

format: **FRA(aexp)**

where:

aexp is an arithmetic expression, variable or constant.

The FRA function returns the FRActional portion of aexp.

Example:

```
>>List

   10       Number=3.7
   20       Print Fra(Number)
   30       End

>>Run
0.7
***30 End***
```

function: **Integer Portion**

format: **INT(aexp)**

where:

aexp is an arithmetic expression, variable or constant.

The INT function returns the largest INTeger value which is less than or equal to aexp.

Example:

```
>>List

  100       Boxes=5.7
  200       Print Int(Boxes)
  300       End

>>Run
5
***300 End***
>>Print Int(-5.7)
-6
```

function: **Integer Random Number Generator**

format: **IRN(X)**

where:

X is a dummy argument.

The IRN function generates an Integer Random Number between 0 and +32767.

Note:

1. To change the sequence of random numbers, the RANDOMIZE instruction should be included in the program.

Example:

```
>>List

   10      Rem Demo of Integer Random Number Generator
   20        For Section=1 to 10
   30        Print Irn(6)
   40        Next Section
   50      End

>>Run
29284
25801
18835
4647
9295
18846
4924
10105
20210
7652
***40 End***
```

The program listed above will print out 10 integer random numbers. If this program is run a second time, it will generate the same 10 random numbers. A different set of random numbers will be generated only if a RANDOMIZE statement is included at the beginning of the program.

function: **Logarithm**

format: **LOG(aexp)**

where:

aexp is an arithmetic expression, variable or constant.

The LOG function calculates the natural LOGarithm (i.e., log to the base e) of aexp.

Note:

1. The logarithm base 10 for X can be computed as follows:

```
Log'base'10 = Log(X)/Log(10)
```

Example:

```
>>List

   10       Input Number
   20       Nat'log=Log(Number)
   30       Print Nat'log
   40       End

>>Run
? 3.2
1.1631508098056
***40 End***
```

function: **Maximum Value**

format: **MAX(aexp1,...,aexpn)**

where:

aexp1-n are arithmetic expressions, variables, or constants.

The MAX function examines the list of arithmetic expressions (aexp1 through aexpn) and returns the value of the largest expression.

Example:

```
>>List

   10      Num1=10
   20      Num2=25
   30      Max'val=Max(Num1,Num2)
   40      Print Max'val
   50      End

>>Run
25
***50 End***
```

function: **Minimum Value**

format: **MIN(aexp1,...,aexpn)**

where:

aexp1-n are arithmetic expressions, variables, or constants.

The MIN function examines the list of arithmetic expressions (aexp1 through aexpn) and returns the value of the smallest expression.

Example:

```
>>List

   10       Num1=5
   20       Num2=10
   30       Min'val=Min(Num1,Num2)
   40       Print Min'val
   50       End

>>Run
5
***50 End***
```

instruction: **Randomize**

format: **[Ln] RANDOMIZE**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The RANDOMIZE instruction is used to reset the random number dummy variable used by the RND and IRN functions so that a different sequence of random numbers will be produced each time the RND and IRN functions are used.

Notes:

1. RANDOMIZE should be used only once within a program utilizing RND to ensure that a truly random sequence of numbers results.

2. RANDOMIZE should be used after generating every 1000 numbers using IRN. This will ensure a truly random sequence of numbers.

Example:

```
>>List

   10      Print "This is a Random number ";
   20      Randomize
   30      Print Rnd(0)
   40      End

>>Run
This is a Random number 0.7137712225
***40 End***
>>Run
This is a Random number 0.8171978025
***40 End***
```

This program will PRINT a different random number every time it is RUN.

function: **Random Number Generator**

format: **RND(X)**

where:

X is a dummy argument.

The RND function generates a RaNDom number in the range $0 \leq Rnd(0) < 1$.

Example:

```
>>List

   10     Rem Demo of Random Number Generator
   20       For Index=1 To 100
   30       Print Rnd(2)
   40       Next Index
   50     End
```

The above example will PRINT 100 random numbers. If this program is RUN a second time, it will generate the same 100 random numbers. To generate a new set of random numbers each time the program is RUN, the above example can be rewritten as follows:

```
>>List

   10     Rem Demo of Random Number Generator
   15     Randomize
   20       For Index=1 To 100
   30       Print Rnd(2)
   40       Next Index
   50     End
```

function: **Sign**

format: **SGN(aexp)**

where:

aexp is an arithmetic expression, variable or constant.

The SGN function returns a +1 if the value of the expression aexp is greater than 0, a 0 if aexp equals 0, and a -1 if aexp is less than 0.

Example:

```
>>List

   10      Input Value1,Value2,Value3
   20      Print Sgn(Value1)
   30      Print Sgn(Value2)
   40      Print Sgn(Value3)
   50      End

>>Run
? -12,0,14
-1
0
1
***50 End***
```

function: **Square Root**

format: **SQR(aexp)**

where:

aexp is an arithmetic expression, variable or constant.

The SQR function calculates the SQuare Root of the positive expression aexp.

Example:

```
>>Print Sqr(9)
3
```

TRIGONOMETRIC FUNCTIONS

Structured Basic offers these trigonometric functions:

Trigonometric Functions

Atn(X)	arctangent of X
Cos(X)	cosine of X
Sin(X)	sine of X
Tan(X)	tangent of X

function: **Arctangent**

format: **ATN(aexp)**

where:

aexp is an arithmetic expression, variable or constant.

The ATN function calculates the ArcTaNgent of aexp.

Note:

1. Although Structured Basic does not include pre-defined Arcsin and Arccos functions, the user can calculate these using the ATN function as follows:

```
Arcsin(X) = Atn(X/Sqr(-X*X+1))
Arccos(X) = -Atn(X/Sqr(-X*X+1))+2. *Atn(1.)
```

Example:

```
>>Print Atn(.80)
0.67474094222353
```

In this example the ArcTaNgent of 0.80 is equal to 0.6747094222353 radians. If the DEGree mode is selected, the arctangent of 0.80 is given in DEGrees:

```
>>Deg

>>Print Atn(.80)
38.659808254087
```

function: **Cosine**

format: **COS(aexp)**

where:

aexp is an arithmetic expression, variable or constant.

The COS function calculates the COSine of the angle represented by aexp.

Notes:

1. See ATN for a description of how to calculate Arccos.

2. Unless DEG mode has been selected, it is assumed that the value of aexp is expressed in RADians.

Example:

```
>>List

   10      Input Num1
   20      Let Num2=Num1*2
   30      Print Cos(Num2)
   40      End

>>Run
? .60
0.36235775447529
***40 End***
```

In this example, the COSine of a 1.20 radian angle is 0.36235775447529.

function: **Sine**

format: **SIN(aexp)**

where:

aexp is an arithmetic expression, variable or constant.

The SIN function calculates the SINe of an angle represented by aexp.

Notes:

1. See ATN for a description of how to calculate Arcsin.

2. It is assumed that the value of aexp is expressed in RADians unless the DEG mode has been specified.

Example:

```
>>List

   10      Input Number
   20      Print Sin(Number*3)
   30      End

>>Run
? .04
0.11971220728892
***30 End***
```

In this example, the SINe of a 0.12 radian angle is approximately 0.12.

In the following example, we first select the DEG mode and then request the program to PRINT the SINe of a 90 degree angle:

```
>>List

    5      Deg
   10      Print Sin(90)
   20      End

>>Run
1
***20 End***
```

The SINe of a 90 degree angle is 1.0.

function: **Tangent**

format: **TAN(aexp)**

where:

aexp is an arithmetic expression, variable or constant.

The TAN function calculates the TANgent of the angle represented by aexp.

Note:

1. It is assumed that the value of aexp is expressed in RADians unless the DEGree mode has been specified.

Example:

```
>>List

    5       Deg
   10       Print Tan(30)
   20       End

>>Run
0.577350269189
***20 End***
```

STRING FUNCTIONS

Structured Basic offers these string functions:

String Functions

Asc(X$)	provides equivalent ASCII numeric value of the first character of X$
Chr$(X)	gives a single character string which is the ASCII equivalent of X
Expand X$,n	inserts n null characters in string X$
Hex$(X)	returns the 4 byte ASCII hexadecimal representation of a number
Len(X$)	returns the length of string X$
Pos(X$,Y$,n)	returns the location of substring Y$ within string X$ starting with character n
Str$(X)	returns the character representation of any numeric expression X$
Val(X$)	returns the numeric representation of any string expression X$.
Valc(X$)	returns the numeric representation of any string expression X$. Sets an error condition for unacceptable values of X$.

function: **ASCII Value of a Character**

format: **ASC(svar)**

where:

svar is a string variable or literal.

The ASC function returns the ASCII decimal value of the first character of string.

Note:

1. Refer to Appendix B for a table of ASCII characters and their values.

Example:

```
>>List

   10      Input Text$
   20      Print Asc(Text$)
   30      End

>>Run
? A
65
***30 End***
```

In this example, the ASCII decimal value for the first character of the string variable Text$ (character A) is 65.

function: **Character**

format: **CHR$(aexp)**

where:

aexp is an arithmetic expression, variable, or constant.

The CHR$ function returns the single CHaRacter which is represented in ASCII by aexp.

Notes:

1. If aexp is outside of the range $0 \leq aexp \leq 255$ an error will be generated.

2. This function allows the user to draw graphs or figures with special characters. The function may also be used to initiate special functions such as cursor positioning, generating line or form feeds, or causing a bell to sound on the terminal.

3. This function may be used to output non-printing and special purpose characters such as control or underline characters.

4. CHR$ may be used any place a string is required.

Example:

```
>>List

   10      Input Value
   20      Print Chr$(Value)
   30      End

>>Run
? 42
*
***30 End***
```

The ASCII decimal value 42 is equivalent to the character *. Thus, in the above example, the instruction PRINT CHR$(42) instructs the computer to output the character * on the terminal.

instruction: **Expand String**

format: **[Ln] EXPAND svar, exp-2**

[Ln] EXPAND svar(exp-1), exp-2

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
svar	is a string variable.
exp-1	is an optional arithmetic expression, variable, or constant. The default value is 0.
exp-2	is an arithmetic expression, variable, or constant.

The EXPAND instruction inserts null characters into a string variable. The number of nulls to be inserted is specified by exp-2. The nulls are inserted before the character specified by exp-1, or, if exp-1 is omitted, before the first character of the string.

Notes:

1. Remember that the first character in a string is located at index position 0 (svar(0)).

2. This instruction is very useful for inserting characters into the middle of a string.

Example:

```
>>List

  100     Word$ = "ABEF"
  110     Expand Word$(2),2
  120     Word$(2,3) = "CD"
  130     Print Word$

>>Run
ABCDEF
```

function: **ASCII Hex Representation**

format: **HEX$(avar)**

where:

avar is an arithmetic variable or constant.

The HEX$ function will convert avar into its 4 byte Hexadecimal ASCII (string) representation.

Example:

```
>>List

  100       Number=32
  200       String$=Hex$(Number)
  300       Print String$
  400       End

>>Run
0020
***400 End***
```

function: **Length of String**

format: **LEN(svar)**

where:

svar is a string variable or literal.

The LEN function returns an integer value which is equal to the number of characters in any string variable svar. In other words, the LEN function gives the LENgth of a string.

Note:

1. Both characters and spaces are counted as part of the length. Trailing null characters are not counted as part of the string length.

Example:

```
>>List

   10      Dim String1$(20),String2$(30)
   20      Input String1$,String2$
   30      @"Length of String1 is ";Len(String1$)
   40      @"Length of String2 is ";Len(String2$)
   50      End

>>Run
? Example
?? Length Command
Length of String1 is 7
Length of String2 is 14
***50 End***
```

function: **Position of Substring**

format: **POS(svar-1,svar-2,aexp)**

where:

svar-1	is a string variable or literal
svar-2	is a substring variable or literal
aexp	is an arithmetic expression, variable, or constant

The POS function is used to locate the POSition of a substring (svar-2) within a string (svar-1). The position within the string svar-1 at which the search is to begin is specified by the arithmetic expression aexp. This function returns a value equal to the POSition index of the first character of the substring within the string.

Notes:

1. A -1 is returned if the substring is not found.

2. Remember that the first element of a string is numbered 0.

Example:

```
>>List

   10      Dim String$(50)
   20      String$="This is a substring search"
   30      String'is=Pos(String$,"is",4)
   40      String'r=Pos(String$,"r",20)
   50      Print String'is
   60      Print String'r
   70      End

>>Run
5
23
***70 End***
```

In this example, the computer is first instructed to search for "is" starting from the fourth character in string$ and second to search for "r" starting from the twentieth character in string$. Starting from position 4, the first character in substring "is" is located in position 5. Starting from position 20, the first character "r" is located in position 23. Consequently, the computer returns a value of 5 for String'is and 23 for String'r.

An application of this function is the conversion of numbers from hexadecimal to decimal. The following program demonstrates the principle using a one digit number:

```
>>List

 1000    Dim Hex'number$(0), Answer$(0)
 1100 *Start : Input "One digit hex number:  ",Hex'number
 1200    Decimal'number = Pos ("0123456789ABCDEF", Hex'number, 0)
 1300    IF Decimal'number = -1 Then Goto Start
 1400    Print "The decimal equivalent is:  ";Decimal'number
 1500    Input "Another one (Y/N)?  ",Answer$
 1600    If Answer$ = "y" then goto Start
 1700    If Answer$ = "Y" then goto Start
 1800    End
```

function: **String Equivalent**

format: **STR$(aexp)**

where:

aexp is an arithmetic expression, variable, or constant.

The STR$ function converts an expression (aexp) to a STRing which is the ASCII representation of the expression.

Notes:

1. Valid input to this function includes the decimal point (.) and a leading plus (+) or minus (-) sign.

2. STR$ may be used any place a string is required.

Example:

```
>>List

   10      Input Number
   20      String$=Str$(Number)
   30      Print String$
   40      End

>>Run
? 8.45
8.45
***40 End***
```

function: **Value of String**

format: **VAL(svar)**

where:

svar is a string variable or literal.

The VAL function converts a string (svar) into a numeric variable which may be used in an arithmetic expression.

Notes:

1. If the argument string for VAL consists of both numeric and non- numeric information, the following conventions hold:

   a. If the first character is non-numeric, VAL will return a zero value (this can be used to decode a user's input). The first character is considered to be numeric if it is the leading percent sign (%) of a hexadecimal constant.

   b. If the first character or characters are numeric they will be converted without consideration of the portion of the string including and following the first non-numeric character.

Example:

```
>>List

   10      String$="26.6321"
   20      Increment=1
   30      Value=Increment+Val(String$)
   40      Print Value
   50      End

>>Run
27.6321
***50 End***
```

function: **Value of String With Error Checking**

format: **VALC(svar)**

where:

svar is a string variable or literal.

The VALC function returns a value which can be assigned to a numeric variable. This function provides a user trappable error when its argument is not properly structured.

Notes:

1. The error conditions that may be set are as follows:

   a. The actual parameter of a VALC function call has evaluated to a numeric quantity whose accuracy is outside the range of Long Floating Point numbers.

   b. The actual parameter of a VALC function call has evaluated to a numeric quantity whose value is outside the range of Long Floating Point numbers.

   c. The actual parameter of a VALC function call was not structured according to the proper syntax for numeric constants.

      The characteristic aspects of this syntactic specification are as follows:

      1. The comma is not allowed to exist in numbers.

      2. If the string contains a decimal point there must be digits on both sides of it.

      3. Both integers and floating point numbers may contain exponents.

      4. Multiple positive or negative signs are illegal.

2. If the process of converting the string is successful, the function will return a numeric value which is assignable to a numeric data type. If the string is properly structured, but evaluates to a value outside of the accuracy (condition A) or range (condition B) of the Long Floating Point numbers, then a run-time trappable error will result. The function may also be aborted for a syntax error (condition C). If it is aborted for any reason, the returned value is undefined.

   If the value is to be assigned to a variable of a less precise data type, the value will be rounded as per Basic's implicit type conversion.

TIME AND DATE FUNCTIONS

Structured Basic offers these time and date functions:

Time and Date Functions

Time$("")	read the time
Time$("hhmmss")	set the time
Date$("")	read the date
Date$("yymmdd")	set the date

function: **Set Time or Read Time**

format: **TIME$("")**
TIME$("hhmmss")

where:

""	has the operating system return the time in the form hhmmss, where hh is the hour, mm is the minutes, and ss is the seconds.
"hhmmss"	sets or resets the time in the operating system, where hh is the hour, mm is the minutes, and ss is the seconds.

The TIME$ function either returns the time or sets the time depending on the parameters used.

Notes:

1. The time may be assigned to a string variable, printed or called in any manner a standard function may be called.

Example:

```
>>List
10 Dummy$=Time$("134700")
20 A$=Time$("")
30 Print A$
>>Run
134700
***End***
```

function: **Set Date or Read Date**

format: **DATE$("")**
DATE$("yymmdd")

where:

""	has the operating system return the date in the form yymmdd, where yy is the year, mm is the month, and dd is the day of the month.
"yymmdd"	sets or resets the date in the operating system, where yy is the year, mm is the month, and dd is the day of the month.

The DATE$ function either returns the date or sets the date depending on the parameters used.

Notes:

1. The date may be assigned to a string variable, printed, or called in any manner a standard function may be called.

Example:

```
>>List
10 Dummy$=Date$("820716")
20 A$=Date$("")
30 Print A$
>>Run
820716
***End***
```

Chapter 17

SYSTEM AND FILE STATUS

This chapter describes the instructions used for controlling the status of the system and of files.

command: **Disk Drive**

format: **DSK**
DSK "X"

where:

X is a disk or directory specifier. Under the CDOS Operating System, the DSK instruction uses the parameter to change the default disk drive to the drive specified. Under the Cromix Operating System, the DSK instruction uses the parameter to change the default directory to a directory of the same name as the specifier.

Under the CDOS Operating System, the DSK instruction displays or alters the default disk drive. Under the Cromix Operating System, the DSK instruction alters the current directory.

Notes:

1. When no parameter is given, DSK displays the current disk drive identifier. Under the Cromix Operating System, the identifier refers to the current directory.

2. Use IOSTAT(0,0) to determine the current drive while in the RUN mode.

3. When DSK is used under the Cromix Operating System, the drive specifier refers to a directory that has the same name as the specified drive. In this case, any reference to drive A automatically refers to the current directory regardless of the name of that directory. Specifying any other directory changes the default directory to a directory that has the same name as the drive specified. A change in the directory remains in effect only during the Structured Basic session. Once the user exits from Basic, the current directory is what it was when Basic began executing.

instruction: **Enable Echo**

format: **[Ln] ECHO**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run-time. Otherwise it is executed immediately.

The ECHO instruction is used to re-enable the display of certain information at the console terminal after the display has been disabled by the NOECHO instruction.

instruction: **Disable Echo**

format: **[Ln] NOECHO**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The NOECHO instruction is used to disable the display of user entered responses to the INPUT instruction.

Notes:

1. This instruction is useful when a secret code or password is to be INPUT. The code will not be displayed on the screen, making theft of the code difficult.

2. The NOECHO mode is <u>not</u> reset by the RUN or SCRatch instructions. It is reset by the ECHO instruction.

Example:

```
>>List

  100       Noecho
  200       Input Secret'number
  300       Print Secret'number
  400       End

>>Run
?           (user entered INPUT is not displayed)
517         (the value of Secret'number is PRINTed)
***400 End***
```

instruction: **Enable Escape**

format: **[Ln] ESC**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The ESC instruction is used to re-enable ESCAPE key operation after it has been disabled by the NOESC instruction.

instruction: **Disable Escape**

format: **[Ln] NOESC**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The NOESCape instruction disables the console terminal ESCAPE key operation.

Notes:

1. Most terminal keyboards include a key labeled ESCAPE. Basic recognizes the escape character as a signal to abort program execution and return to the command mode. The NOESC instruction is used to prevent program interruption when the ESCAPE key is pressed.

2. ESC results from the use of the CONTROL-Z, ESCAPE, or CONTROL-[key.

3. ESCAPE key operation is reset by the ESC instruction.

4. Most programs will execute significantly faster when the operation of the ESCAPE key has been disabled by the NOESC instruction.

function: **Free Space**

format: **FRE(X)**

where:

X is a dummy argument.

The FRE function gives the number of bytes of memory in the User Area which are currently FREe or unused. It is a long floating point value.

Notes:

1. Certain statements do not occupy their full space until after they have been executed (e.g., DIM).

2. Because Basic allocates space for its internal tables in segments, the FRE function is only an approximation of the actual number of bytes available to the user.

3. Space recovered from DELETEd lines is available for new program lines, not for arrays or variables. Thus, the FRE function will not necessarily reflect available program space.

   If statement lines have been DELETEd from the User Area, it will be necessary to perform the following steps in order for the FRE function to reflect the change:

   a. LIST the program (do <u>not</u> SAVE it) to a temporary disk file.

   b. SCRatch the User Area.

   c. ENTER the temporary disk file.

   d. Give the command PRINT FRE(X).

function: **I/O Status**

format: **[Ln] IOSTAT(aexp1, aexp2)**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

aexp1 is a channel number.

aexp2 is the status parameter.

The IOSTAT function returns the current status of an OPENed Basic file.

Status Parameter	Returned Value	Meaning
0	XX	Latest error code when reading. See Cromix manual for error codes.
1	X	Sector number for file pointer.
2	Y	Byte number in sector X (above).
3	Z	File pointer, long floating point.

Notes:

1. Devices other than disk drives may or may not return status values.

2. If Rec is the record size and File is the file number, the expression:

```
INT((128.0 * Iostat(File,1) + Iostat(File,2))/Rec)
```

will give the current record number. A similar expression can derive the current byte within the record.

instruction: **On Error Transfer Control**

format: **[Ln] ON ERROR STOP**

[Ln] ON ERROR GOTO n

[Ln] ON ERROR GOSUB n

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n — is the line number or line name of the statement to which control is transferred.

The ON ERROR instruction causes program control to be transferred as specified (STOP, GOTO, or GOSUB) when a non-fatal error occurs during program execution.

Notes:

1. A non-fatal error in Basic is any error listed in the error table (see Chapter 24) with a number of 128 or greater. Errors numbered 127 and below are defined as fatal errors and cannot be trapped with an ON ERROR statement.

2. If ON ERROR is written at the beginning of a program, the instruction specified with ON ERROR will be executed each time a program error occurs. If placed elsewhere in the program, the instruction will be executed only for errors which occur during the execution of statements following the execution of the ON ERROR statement.

3. ON ERROR is reset by the RUN, SCRatch, and ON ERROR STOP instructions.

4. Using RETRY to exit from a subroutine will cause the instruction which caused the error to be executed again. Refer to the RETRY instruction.

Example:

```
 60    Input Num1,Num2
 80    Print Num1*Num2
100    On Error Goto 300
120    Input Num3,Num4
140    Print Num3/Num4
160    Goto 60
300    Print "A non-fatal error has occurred"
320    Goto 120
```

In this example, any error which occurs before line 100 has been executed for the first time will be dealt with by the standard system error handling procedure. Any trappable error which occurs after line 100 has been executed will cause program execution to continue with statement line 300.

instruction: **On Escape Transfer Control**

format: **[Ln] ON ESCAPE STOP**

[Ln] ON ESCAPE GOTO n

[Ln] ON ESCAPE GOSUB n

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n — is the line number or line name of the statement to which control is transferred.

The ON ESCAPE instruction causes program control to be transferred as specified (STOP, GOTO, or GOSUB) when the ESCAPE key is depressed during program execution.

Notes:

1. If ON ESCAPE is written at the beginning of a program, the instruction specified with ON ESCAPE will be executed each time the ESCAPE key is depressed. If placed elsewhere in the program, the statement will be executed only when the ESCAPE key is depressed following the execution of the ON ESCAPE statement.

2. ON ESCAPE is reset by the RUN, SCRatch, and ON ESCAPE STOP instructions.

Example:

```
>>List

   10      *Begin : Input Cond1,Cond2,Cond3
   20        If Cond1 = -1 Then 70
   30        Let Answer=Cond1+Cond2+Cond3
   40        On Esc Goto Begin
   50        Print Answer
   60      *Wait : Goto Wait
   70        End

>>Run
? 10,15,20
45              (ESCAPE must be pressed here to get out of the Wait
                loop.)
? 70,10,5
85
? -1,0,0
***70 End***
```

instruction: **Set System Parameter**

format: **[Ln] SET aexp1, aexp2**

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
aexp1,2	are arithmetic expressions, variables, or constants.
aexp1	is the system parameter number.
aexp2	is the value to be assigned to the parameter.

The SET instruction is used to change the value of a given system parameter.

Notes:

1. Refer to the SYS instruction for a list of all user accessible system parameters.

2. System parameter number 0 is the width of a page in characters and may assume any value between 0 and 32767. The default value for this parameter is 80 characters which corresponds to a standard 8 1/2 inch page.

   A special use of the SET command is:

   ```
   Set 0, -1
   ```

   which inhibits the automatic carriage RETURN-LINE FEED at the end of a line when page width is reached. Refer to the discussion of the PRINT instruction as it pertains to file output.

   Using SET 0,-1 to disable page width checking is especially useful for graphics output to devices such as the Cromemco 3355 printer.

3. System parameter 1 is the tab field width and may assume any value between 1 and 32767. This corresponds to the width of the field which is output by a comma (,) in a PRINT instruction. If the tab field width is SET to a 0 or -1, no tabbing takes place (i.e., commas are treated as semicolons). The default value for this parameter is 20 characters.

4. System parameter 5 may be used to facilitate timed input. To start the timer, the SET 5 instruction is given, with aexp2 equal to approximately ten (10) times the number of seconds of delay desired. For example:

```
Set 5,50
```

will allow about 5 seconds for a complete user response.

When the next INPUT instruction is encountered, Basic will issue an Error 210 -- INPUT Timeout if the user does not respond with a complete INPUT within the allotted time.

Once SET, the time used to respond to all subsequent INPUT instructions is added together until it exceeds the specified value, and then an error message is generated. It is therefore necessary to give the SET instruction prior to each timed INPUT. When timed INPUT is no longer desired, it may be de-activated by coding:

```
Set 5,0
```

The ON ERROR statement can be used to trap the timeout error. The programmer may find out how much time was used by coding SYS(5) to find the time remaining.

function: **System Parameter**

format: **SYS(aexp)**

where:

aexp is an arithmetic expression, variable, or constant.

The SYS function provides system information based on the value of the argument aexp.

Notes:

1. See the SET instruction for methods of changing the values of these system parameters.

Example:

```
100       On Error Goto Error'routine
          .
          .
140       Open \1\ "FILE.DAT"
          .
          .
699       *Error'routine
700       If Sys(3)=128 Then Create"FILE.DAT":Goto 140
          .
          .
```

In this example, if the file FILE.DAT did not originally exist on the disk, the error routine at line 700 would CREATE the file. Lines 710, 720 etc. could be written to test for other values of SYS(3), and take appropriate action(s).

The table on the following page lists all of the user accessible system parameters.

The first column represents the Value (aexp) of the System Parameter. SET indicates if it is valid to SET this parameter (Yes) or if it may just be examined by the SYS function (No). The Range and Default are those of the Parameter while the last column is a brief description of the Parameter.

Value	SET	Range	Default	System Parameter
0	Yes	≥0	80	Page width
1	Yes	≥0	20	Tab field width
2	No	0-255	-	character last printed
3	Yes	0-255	-	Last run time error
4	Yes	0-Sys(0)	-	Current print column
5	Yes	any	0(off)	INPUT timeout; value 14 must be set to 1 to use this
6	Yes	0,1	0	Upper case LISTing
7	Yes	0,7	0	LIST constants with spaces
8	Yes	-2 thru 5	0	Indentation control
9	Yes	0,1	0	Ignore quotation marks on INPUT
10	No	0-7	-	Current Partition number
11	Yes	1-15	4	Basic-KSAM Pages per Data Block
12	Yes	1-15	4	Basic-KSAM Pages per Key Block
13	reserved			
14	No	0,1	0	Sets Console device driver into character mode
15	Yes	any	-	Reserved for user inter-program communication
16	Yes	0,1	0	0 sets integers to Z80 order for PUTSs/GETs (low, high). 1 sets integers to 68000 order for PUTs/GETs (high, low).
17	reserved			
18	reserved			
19	reserved			

(Rest use greater than 16-bit values:)

100	No	Address space	Gives the address of a 256-byte patch area near the beginning of sbasic60.bin after the version number. This area can be dynamically patched during run-time by means of POKE or permanently patched with the Cromix patch utility.
101	No	Line number	Gives line number of the statement that was executed when an error occurred when ON-ERROR was active.
102	reserved		

function: **Execute a Shell Command**

format: **[Ln] sh**
[Ln] sh svar

where:

Ln is an optional line. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The sh command causes program control to be suspended and transferred to a Cromix shell. The Cromix exit command will return control to Basic.

If a command-line is specified as the svar, it will be executed by the Cromix shell and control then returned to Basic.

Chapter 18

MACHINE LEVEL INSTRUCTIONS

This chapter describes the instructions used for using the system on a machine level and for executing machine level routines.

function: **Address Of A Variable**

format: **ADR(var)**

where:

var is a string or arithmetic variable.

The ADR function will return the memory storage location starting address of a variable as an integer.

Notes:

1. The ADR function allows the use of subscripted variables.

function: **Input From I/O Port**

format: **INP(m)**

where:

m is an I/O port.

The INP function is used to read the contents of INPut port (m).

Note:

1. The value of m must be in the range $0 \leq m \leq 255$.

Example:

```
 10    Noesc : Integer Character
 20    *Poll : If Binand(Inp(0),%0040%) = 0 Then Goto Poll
 30    Character = Inp(1)
 40    Character = Binand(Character,%007F%)
 60    If Character = %001B% Then Print : Goto Escape
 70    Print Chr$(Character);
 80    If Character = 13 Then Print
 90    Goto Poll
100    *Escape : Esc
110    End
```

When a character is typed at the console, this program will echo it back to the console CRT. A RETURN will be echoed as a RETURN followed by a LINE FEED, and an escape character will terminate the program.

The NOESC instruction at line 10 is necessary to disable Basic's continuous polling of input port 0. While a program is being executed, Basic ignores most characters typed at the console. Basic continually polls the UART status port 0 and if a character is ready, reads it from port 1. If this character is not an escape character, it is ignored by Basic. If it is an escape character, program execution is terminated.

Upon reading the UART status, the status bit is reset. It is necessary to disable Basic's polling of the status port so that this user program will be able to determine if a character has been typed. If this were not done, Basic would, by reading the status port, reset the character ready bit, thereby preventing the user program from determining that a character had been typed.

Line 60 replaces the ESCAPE key function by testing the character typed by the user. If it is an escape character, program execution is terminated.

Statement 20 is a loop which continually polls the UART status ready port to determine if a character has been typed. The Receiver Data Available flag is bit 6 (refer to the 4FDC manual). The contents of input port 0 is ANDed with 40 hex to determine if this bit is a one or a zero. If it is a zero, no data is available, and control remains on line 20, polling the input port again.

If the bit has been set (=1) it indicates that data is available, and statement 30 reads the data from input port 1 and assigns this data to the variable Character. Line 40 strips the parity bit off of the input data (the ASCII character set requires only the seven bits, 0 through 6) and as stated before, line 60 terminates program execution upon determining that an escape character (1B hex) has been INPut.

The rest of the program prints the ASCII character whose value was INPut, PRINTs a LINE FEED if a RETURN was INPut, and transfers control to line 20.

When an escape character is detected, control is transferred to line 100 which re-enables the operation of the ESCAPE key and terminates program execution.

instruction: **Output To I/O Port**

format: **[Ln] OUT m,b**

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
m	is an I/O port
b	is a byte value

The OUT instruction is used to OUTput data byte (b) to OUTput port(m).

Notes:

1. The value of m must be in the range $0 \leq m \leq 255$.

2. The value of b must be in the range $0 \leq b \leq 255$.

Example:

```
>>Out 1,75
```

will display the character K (ASCII 75) on the console terminal (output port 1).

function: **Peek At Memory**

format: **PEEK(m)**

where:

m is a memory location (long value).

The PEEK function returns the contents of memory location (m).

Note:

1. Hexadecimal numbers may be used to access locations in the range: $0 \leq h \leq$ %FFFF%. Negative decimal numbers map into their hexadecimal equivalents:

```
Peek(-2)
```

is the same as

```
Peek(%FFFE%)
```

Example:

```
Print Peek(5)
```

will PRINT the contents of memory location 5.

function: **Poke Into Memory**

format: **[Ln] POKE m,b**

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
m	is a memory location (long value).
b	is a byte value.

The POKE instruction puts the byte value b into memory location m.

Notes:

1. Hexadecimal numbers may be used to access locations in the range: $0 \leq h \leq$ %FFFF%. Negative decimal numbers map into their hexadecimal equivalents:

```
Poke -2,5
```

is the same as

```
Poke %FFFE%,5
```

2. The value of b must be in the range $0 \leq b \leq 255$.

Example:

```
Poke 1000,255
```

will place the value 255 into memory location 1000.

function: **Call a User Program**

format: **USR(m,P1,...,Pn)**

where:

m is the address of the assembly language routine.

P1-n are parameters that are converted to 32 bit integers.

The USR function makes it possible to call an assembly language subroutine from a Basic program. It turns a long floating point value.

Sbasic68 contains a 256-byte patch space that may be used for USR routines. Use SYS(100) function to get the address of this patch space.

Notes:

1. The USR function always requires the user to specify one parameter in addition to the address, even if it is a dummy parameter. For example, USR(0,1) is correct, while USR(0) will result in a syntax error.

2. If m is decimal, it must correspond to a legal memory address. Hexadecimal numbers may be used to access locations only in the range: $0 \leq h \leq \%FFFF\%$.

3. USR is a function and must take the form of a function when it is used:

```
>>100 Alpha = Usr(X,Y)
```

4. When the user routine gains control (at the address specified in the USR function call), the following conventions apply:

   a. Register D0 contains the number (n) of parameters in the function call.

   b. Register A4 contains the return address to Basic. The user routine may re-enter Basic by coding the following assembly language instruction:

```
JMP (A4)
```

   c. The parameters are placed in order (P1, P2, ..., Pn) on the CPU stack (A7) and may be recovered via the instructions:

```
POP.L     D5
MOVE.L    -(A7), D5
```

d. If and only if n parameters (n is the contents of register D0, as above) are Popped off the stack, and the stack pointer is not otherwise changed, Basic may be re-entered via the following instruction:

```
RTS
```

e. The routine may return a long value to be assigned to the function by placing the value in the D6 register before re-entering Basic.

f. If registers D0 through D4 are used, they must be restored or set to 0 prior to return to Basic.

function: **Type Of Variable**

format: **TYPE(aexp)**

where:

aexp is an arithmetic expression or variable.

The TYPE function will return a value indicating the TYPE of exp.

Note:

The following values are returned by TYPE:

Value	**TYPE of Expression**
1	Integer
2	Short Floating Point
4	Long Floating Point

functions: **Basic-KSAM Numeric Sorting Conversions**

formats: **IKEY$(aexp)**

FKEY$(aexp)

KEYI(svar)

KEYF(svar)

where:

aexp is an arithmetic expression, variable, or constant.

svar is a string variable or string literal.

These four functions allow the user to store numeric information in character variables so that numeric fields of a data file may also be key fields. The conversion is such that the ordering of the records with regard to these key values will be the ordering of the key fields, interpreted as numeric quantities.

Notes:

1. The Ikey$ function will take an integer variable argument and return two character positions of a string variable. When these two character positions are used as a key field, Basic-KSAM will put them into the file in the proper order of the integer argument. Thus, integers may be used as keys, and will be sorted according to the integer value.

2. The Fkey$ function performs the same operation as the Ikey$ function, except it takes a Long Floating Point variable as an argument and produces eight characters of a string variable as a result. These eight characters also cause the record to be added to the Basic-KSAM file in the order of the numeric quantity of the argument of the function. Short Floating Point numbers may be utilized by ignoring the last four bytes returned.

3. The Keyi function performs the opposite operation of the Ikey$ function. When a record is read into a string variable which contains bytes that are really the converted form of an integer, this function will translate them into the integer representation and assign the value to an integer variable. Thus, the argument for this function is two bytes of a string variable and the result is of type integer.

4. The Keyf function performs the reverse of the Fkey$ function, and thus will take from four to eight characters of a string variable as an argument and return a Basic Long Floating Point number which may be assigned to a Long or Short Floating Point variable.

Chapter 19

SCOPE OF VARIABLES

This chapter describes the instructions used for setting up COMMON and LOCAL variables.

statement: **Common Storage Area Method I**

format: **Ln COMMON**

where:

Ln is a line number.

This version of the COMMON statement may be used when chaining programs (using RUN statements in the program). Refer also to the BEGINCOMMON and ENDCOMMON instructions in the description of Common Storage Area Method II.

This statement reserves a Common Storage Area for variable storage. The size and contents of this area are determined by the string variables and array variables which have been explicitly DIMensioned (used in a DIM, INTEGER, SHORT, or LONG instruction) since the last RUN instruction or since Basic was loaded. The space is reserved in a byte-by-byte fashion without regard to variable type or length.

A subsequent RUN instruction will initialize all variables except those in the Common Storage Area. The reserved area will be assigned on a byte-by-byte basis to variables as they are DIMensioned in the new program (the program called by the RUN instruction).

The Common Storage Area will cease to be reserved if two RUN instructions are issued without an intervening COMMON statement.

Notes:

1. It is up to the user to ensure that DIMensioned variable types match between programs. For example, if program A is as follows:

```
200     Dim Name$(25), Interest'rate(10,2)
300     Integer Time'period(100)
400     Common
500     Run "B"
```

then program B, if it has explicitly DIMensioned variables, must DIMension:

a. 26 bytes of string (ASCII) variables

b. 264 bytes of Long Floating Point variables (8 bytes each)

c. 202 bytes of Integer variables (2 bytes each)

in that order. The names and lengths of the strings and arrays do not matter, as long as the types match. Program B could use either of the following sets of statements:

a.
```
10 Dim Title$(9), Reference$(15), Table(10,2)
20 Integer Time'prior(25), Time'current(24)
30 Integer Time'future(49)
40 Common
```

b.
```
10 Dim Last'name$(1), First'name$(1), Middle'name$(1)
20 Dim Initials$(19), Value(10,2)
30 Integer Buffer(100)
40 Common
```

Notice that the 27th through 290th bytes in the reserved area must be DIMensioned similarly in each program if the same indices are to be used in each program. This is true for all two and three DIMensional arrays.

statement: **Common Storage Area Method II**

format: **Ln BEGINCOMMON**

Ln ENDCOMMON

where:

Ln is a line number.

This version of the Common statement may be used to establish a Common Storage Area when using Procedures.

A Common Storage Area is provided so that the programmer may declare certain strings and arrays as global throughout a series of calling and called Procedures.

A global string or matrix variable is one which may be referenced from a Partition other than the one in which it was originally declared. The value of a global variable may be altered both by the calling as well as the called Procedure regardless of whether the Procedures reside in the same or different Partitions.

Variables which are not used as receiving parameters in a procedure are always global within one Partition.

Defining and Accessing the Common Storage Area

Only matrices (1, 2, or 3 DIMensional) and string variables can be placed in the Common Storage Area.

The amount of Common Storage Area which is established by the main program determines the maximum amount of Common Storage Area for all called Procedures.

When the RUN command is given, before any Basic instruction is executed, there is an implicit BEGINCOMMON instruction.

If the user does not wish to have a Common Storage Area, an ENDCOMMON instruction should be coded in the main program before any definition of or references to subscripted variables. This will cause Basic to reserve a Common Storage Area of length zero (i.e., no Common Storage Area).

If the user wishes to have a Common Storage Area for string variables and arrays, this area may be reserved by DIMensioning the variables (or by referencing the subscripted variables which implicitly DIMensions them) and then coding the ENDCOMMON instruction in the main program (the program residing in Partition zero).

Note that ENDCOMMON is implicit in a call to a Procedure or a return from a Procedure (ENDPROC or ERRPROC).

Once ENDCOMMON has been executed (either explicitly or implicitly) the Common Storage Area has been reserved. Loading another module or transferring control to another Partition will not alter this area.

It is not possible to expand the Common Storage Area to a size greater than was originally defined in the main program. The size of the Common Storage Area remains fixed until another RUN instruction is executed.

In order to access the Common Storage Area from a called Procedure in another Partition, it is necessary to code the BEGINCOMMON and ENDCOMMON instructions in the called Procedure. In between these two instructions, any definition of subscripted variables will cause the elements of the array or string to reference the Common Storage Area.

Storage is allocated from the bottom of the Common Storage Area, sequentially, as subscripted variables are defined. Note that string variables occupy one byte for each element (character), while elements of an Integer, Short or Long Floating Point array will occupy 2, 4, and 8 bytes respectively. When assigning subscripted variables to the Common Storage Area, it is the programmer's responsibility to ensure that each type of variable is assigned to a sequence of bytes which were originally that same type of variable.

Remember that all arrays start with element zero (i.e., an array which is DIMensioned as 20 contains 21 elements numbered 0 through 20).

Example:

```
>>Run
This is a message
5          10                    15
***240 End***
>>List

      110    Rem MAIN Program
      120    Rem
      130    Rem
      140    Rem There is an implicit BEGINCOMMON instruction
      150    Rem before the first program instruction.
      160    Lib"comexamp"
      170    Dim Strg$(19),Var(2)
      175    Endcommon
      180    Strg$="This is a message"
      190    Var(0)=5 :  Var(1)=10 :  Var(2)=15
      210    Rem The following array is NOT in COMMON.
      220    Dim Address(100)
      230    Call .Called'procedure
      240    End

>>Use 7

>>List

      100  Procedure .Called'procedure
      110    Rem
      120    Rem COMMON is not accessed until the BEGINCOMMON
      130    Rem instruction is given.
      140    Begincommon
      150    Dim Message$(19),Table(2)
      160    Endcommon
      170    Print Message$
      180    Print Table(0),Table(1),Table(2)
      190    Endproc
      200    End
```

instruction: **Define Local Variable**

format: **[Ln] LOCAL var-1, Mat var-2,...**

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

var1-n — are scalar, matrix, or string variables. Matrix variables must be preceded by the word MAT.

The LOCAL instruction defines a new variable (or string or explicitly DIMensioned matrix) with the same name as a previously defined variable. The previous value of the variable is saved for later use. This process may be repeated as many times as necessary (or as storage space permits). The previous value of a variable will be recalled when a control structure, which was pending before the LOCAL instruction was given, is completed.

Notes:

1. The control structures which will cause the previous value of a variable to be restored are ENDDO, ENDWHILE, NEXT, ENDPROC, ERRPROC, EXITPROC, RETURN, and UNTIL.

2. DO and ENDDO may be used as a do-nothing control structure to cause variables to be popped as desired.

3. When a variable is defined as LOCAL, the contents of the variable are zeroed (numeric) or set to null characters (string).

4. This explanation may help clarify the LOCAL instruction to those users who are familiar with LIFO stacks. The LOCAL instruction allows the user to push the value(s) of a variable, string, or explicitly DIMensioned matrix onto the run time stack. The value(s) will automatically be popped back into their respective variables upon completion of a pending control structure.

5. When a DIMensioned string or arithmetic variable is defined as LOCAL, it must be re-DIMensioned. The LOCAL variable is a new variable with a value of zero (or null characters in the case of a string) and no specified DIMension.

Examples:

```
100     Length=5
110     Do
120     Local Length
130     Length = 10
140     Print Length
150     Enddo
160     Print Length
170     End

>>Run
10
5
```

In this example, the variable Length is assigned a value of 5 on line 100. A control structure is introduced on line 110. The purpose of the DO-ENDDO control structure is to restore the variable Length to its former value after being used as a LOCAL variable. Line 120 creates a new variable, Length. Length has a value of 0 until, on line 130, it is assigned a value of 10. Line 150 (the termination of a Pending control structure) restores the old value of 5 to Length as can be seen from the PRINT instruction on line 160.

The following example demonstrates a more practical use of the LOCAL instruction. Each time before the recursive Procedure .Factorial calls itself, the variable N is saved (on the run time stack) for later use. Each time .Factorial ends and goes back to a previous level, the variable N is recalled (popped) and used in computing Nfactorial (line 1150).

Refer to the chapter on Procedures for additional information on the use of Procedures and automatic local variables.

```
>>List

  100     Rem Program to compute factorials.
  110     Long Answer :  Integer N,Boogie :  Dim Response$(0)
  120     Library"factor"
  130     Boogie=1
  140       While Boogie=1
  150       Input"number:  ",N
  160       If N>49 Or N<1 Then Do
  170         @"The number cannot be larger than 49,"
  180         @"Nor can it be smaller than 1."
  190         Else
  200         Call .Factorial (N;Answer)
  210         Print N;"!  = ";Answer
  220         Enddo
  230       @
  240       Input"Another?  ",Response$
  250       Rem Y or y says:  calculate another factorial.
  260       If Response$"y" And Response$"Y" Then Boogie=0
  270       Endwhile
  280     End

>>Use 7

>>List

 1000     Procedure .Factorial (N)
 1010       Rem Recursive Procedure for Computing Factorials
 1020       Rem SAVE file name:  FACTOR
 1030       Integer New'n
 1040       If N=1 Then Do
 1050         Rem initialize Nfactorial and End Procedure
 1060         Nfactorial=1
 1070         Else
 1080           Do
 1090           Rem Decrement N and Call Factorial
 1100           New'n=N-1
 1110           Local N
 1120           Call .Factorial (New'n;Nfactorial)
 1130           Enddo
 1140         Rem Compute Nfactorial on the way back
 1150         Nfactorial=N*Nfactorial
 1160         Enddo
 1170       Endproc (Nfactorial)
 1180       End

>>Run "Main"
number:  3
3!  = 6
```

```
Another?  Y
number:  55
The number cannot be larger than 49,
Nor can it be smaller than 1.

Another?  Y
number:  5
5!  = 120

Another?  n
***280 End***
```

Chapter 20

PROCEDURES

A major facet of structured programming is modularization. The implementation of Partitioned memory, Procedures, and a Procedure Library makes Cromemco Basic truly a Structured Basic.

Procedures differ from subroutines mainly in flexibility. A procedure may be stored in a library and loaded just by Calling it. A procedure in any partition may be called from any other partition, while a subroutine must reside in the same partition as the program that uses it. Procedures allow programs that are more modular by passing variables explicitly between program and procedures. Finally, procedures allow the programmer to build large programs in limited memory by loading program segments only as needed and reusing memory occupied by inactive program segments.

Under Structured Basic, the User Area is divided into 8 Partitions which are numbered zero through seven. Until used, each Partition is a null Partition and occupies no space in the User Area. When Basic is loaded, the user has immediate access to Partition zero. This is normally where execution of a program starts and where the main program is stored.

A Procedure is a section of a Basic Program which is designed to do one complete task or an integral subtask. Procedures may call other Procedures. They may have parameters passed to them by the calling routine and also return parameters back to the calling routine. Procedures may be recursive, that is call themselves. (Refer to the LOCAL instruction.)

One or more Procedures may be loaded into each Partition. This may be done manually by the USE and LOAD (or ENTER) instructions, or automatically from a RUNning program by the LIBRARY instruction and Procedure CALLS.

Structured Basic allows the user to build a Procedure Library by means of the Basic Program LIBBUILD. Once this Library has been opened (refer to the LIBRARY instruction), all Procedure CALLs which cannot be resolved by reference to one of the eight Partitions will cause the Library to be searched for the requested Procedure.

If the Procedure is found, it will be loaded into an available Partition (refer to the Procedure CALL instruction).

Unless explicitly requested (by being passed as a parameter or by the use of a Common Storage Area, refer to the BEGINCOMMON and ENDCOMMON instructions), variables are local to a Partition. If more than one Procedure resides within a Partition, the variables are global for all Procedures within

that one Partition. Thus, if Procedures .AAA and .BBB reside in Partition zero and Procedure .ZZZ resides in Partition seven and the variable Time is defined in Procedure .AAA, the variable Time will be accessible from Procedure .BBB (in the same Partition as its definition) but will not be accessible from .ZZZ (in another Partition) unless it is passed in common or as a parameter.

When parameters are passed to a Procedure (refer to the Procedure CALL instruction, parset1) they are automatically declared as local variables within the called Procedure. This is beneficial for calls to other Procedures within a single Partition or for the implementation of recursive Procedures.

USING PARTITIONS

Under Structured Basic the user area of memory is divided into 8 partitions numbered 0 through 7. The main program resides in, and begins execution in, partition 0. Modules are loaded into partitions as called, beginning with the highest available partition on down. Each partition can contain just one module, though a module may contain many procedures.

Partition size depends on the size of user memory and the size of the program segment loaded into it. A partition shrinks or grows to accommodate the program text as necessary, up to the total size of user memory. Procedures are loaded from a library on a space-available basis. If a procedure that is called is not in memory, it is loaded from disk. If there is no space available, any library procedures not currently being executed are overlaid by the called procedure. Thus, the user need not be concerned about the size of an individual partition, only the total memory available for programs.

Variables defined in one partition are not available to other partitions. The variables are local to that partition. However, within the local partition, any procedure or program segment may access any variable that is not explicitly declared as common or passed as an argument to a procedure. Within the local partition, variables are global.

There are two ways to access a partition. Under program control, a partition is accessed when a procedure that resides in that partition is CALLed. When programming, you can enter any partition by executing the command **USE n**, where **n** is the partition number. Once in a partition, any program text can be entered. Normally, however, the main program is entered into partitions 0, while procedures are entered in partitions 1 thru 7.

A partition may be cleared by using the instruction **CLEAR n.** This is the equivalent of SCR but is limited to the partition specified by **n.**

All Basic command mode instructions are local to the current partition. For example, if you are in partition 7, a LIST command will list only the program text in partition 7. The only exceptions are instructions which affect the operating environment (SET, OPEN, CLOSE, etc). The RUN instruction will automatically transfer control to partition 0.

USING PROCEDURES

Procedures are a powerful tool for building structured programs and saving programming labor. They allow a programmer to build and use a library of frequently used procedures.

Once a procedure is stored in a library, it can be included in any program by first specifying the library that the procedure is in and then calling the procedure. If the procedure is not already in memory when it is called, it is loaded from disk. Note that the library must reside either on the current disk or on the disk in drive A. Procedures are loaded into the highest available partition, starting with partition 7 on down.

When a procedure is loaded into a partition, all procedures in the same module are loaded along with the procedure that was called. Thus a group of procedures that function together can be loaded all at once by calling any one of them.

Variables are passed to the procedure via the variable list in the CALL instruction. The actual value of a scalar variable is passed to the receiving variable specified in the procedure definition. This means that although the scalar variable specified in the procedure definition may have the same name as the variable that was passed in the CALL instruction, they are not the same variable. (The name need not be the same.)

Suppose, for example, that the variable named Index is passed through the CALL instruction to .Procedure'1 to the variable in the procedure definition named Index. Index is a separate variable in the calling program and the procedure, but Index (in the procedure) is assigned the value of Index (in the calling program). After the procedure finishes and control has returned to the calling program, examine the value of Index. It will be the same as it was when the procedure was called.

In order to pass the value back from the procedure, Index must be specified as a return argument in both the CALL instruction and the ENDPROC instruction. When the ENDPROC instruction returns control to the calling program, it assigns to Index (in the calling program) the value of Index (in the procedure).

For scalar variables, the type of the destination variable is made to match the type of the variable being passed. For example, the variable A'int is an integer in the calling program. The variable that it is being passed to, A'lfp, is defined as long floating point (lfp). When A'int is passed, A'lfp is converted to integer. Similarly, passing an lfp variable to an integer will result in the type of the receiving variable to be changed to lfp.

Scalar variables that are not used as arguments for a procedure are global within the partition they are defined in. Therefore, they may be modified by any procedure within the partition without being passed. While this may be legal, it is not recommended. It is good programming practice to limit exchange of variables between procedures to explicitly defined arguments. Wholesale use of global variables will only make your programs harder to debug. A good example is a procedure that was entered and debugged in the same partition as

the calling program. When the procedure is later called from a library, it will be loaded in a different partition. Variables which were previously global are not now, and can be a source of mysterious bugs.

For this reason, it is recommended that you: 1) Make a list of variables used in each procedure and not reuse them elsewhere, and 2) Set aside a group of variable names to be used as scratch variables (for instance, all variable names starting with X). These scratch variables may be used freely within any procedure, but should not be relied upon to have the same value the next time the procedure is called. The only variable that should be used as input to a procedure should be one that is passed through CALL arguments.

Matrix and string variables are handled differently. By specifying a matrix or string variable as an argument, you declare it common between the procedure and the calling program. It is the same variable even though it may be referred to by a different name in the procedure than it was in the calling program. Therefore, any value assigned to it by a procedure will be retained when control returns to the calling program.

The use of COMMON in this Basic is different from the COMMON in Fortran or most Basics. Using the COMMON instruction means that all variables explicitly defined before its occurrence are not cleared from memory when another program is Loaded and Run. It does not mean that those variables are available to other partitions. BEGINCOMMON and ENDCOMMON are used to define variables that are available to all partitions. Therefore, the COMMON instruction is used to preserve variables between program overlays.

COMMON string and matrix variables defined in the area between BEGINCOMMON and ENDCOMMON (referred herein as the common area), may be referenced or altered in any other partition as long as they are defined in a common area in that other partition. Due to the method of storing the common variables, it is important that the definitions of the variables are exactly the same in the main program and the procedure.

USING LIBRARIES

A library is a collection of SAVEd basic procedures. To build a library of procedures, each procedure or group of procedures must be SAVEd on the same disk as the library. All procedures that were SAVEd at the same time (are in the same program file) constitute a library module. If the modules you want are not on the correct disk, copy them using the Xfer utility.

Load Sbasic and run LIBBUILD. You must first create the library, then add module files to the library. LIBBUILD creates an index of the library modules and stores it on the disk. The index and the program modules together make up the library.

When you want to call modules in a library, specify the library by the form **LIBRARY svar** where **svar** is a quoted literal or a string variable containing the name of the library file given when it was created. Once the library has been specified, any module in that library may be loaded by CALLing any

procedure in that module. When a procedure is loaded, all other procedures in the same module are loaded at the same time. Therefore, saving unrelated procedures in the same module will create unnecessary overhead at run time.

Once a library has been specified, it is considered the current library, and you may call any procedure in it. To load a module from another library, it is necessary to specify that library as the current library. Once a procedure is loaded, it remains in memory until either SCR, CLEAR, or Bye is executed, or it is overlaid by another procedure.

A caution to users who have defined more file channels than the standard 8. File channel 9 is reserved for the library. Use of file 9 and a library in the same program will result in an Error 131 -- File Already Open message.

Another caution. Do not delete the SAVEd files of procedures that have been put into a library because there is no easy way of getting them out again. This causes problems when a change is necessary to a procedure in the library. If this has happened to you, there is a method for retrieving the module. Clear memory using the SCR instruction. Using command mode, select the library that the procedure is in. Again using command mode, CALL the procedure. Enter partition 7 (where the module will be loaded into) and then SAVE or LIST the procedure to the disk.

Repeat this process for each library module that you want to retrieve, remembering to SCR memory each time. After the changes are made, delete the modules from the library (this doesn't affect the files you just saved) and add the changed versions.

EXAMPLE PROGRAMS

It is recommended that you enter and run the following sample programs yourself. The REM instructions are there to help you understand the programs, but you don't need to enter them. It would also be a good idea for you to experiment with and expand on the programs. Try running procedures in different partitions, saving the procedure in the first program in a library and calling it, etc.

The first program demonstrates the passing of variables to a procedure. Notice that when the value of Index is printed after the return from the procedure, it is not the same as the value that was assigned to Index in the procedure.

Example 1

```
10   Rem      Procedure demonstration - Program 1
20   Rem      Demonstrates locality of variables between procedures
30   Rem      and the root program.
50   Dim String$(20),Array(9)
60   Rem
70   Rem      Assign a value to String$
90   String$="Initial value"
100   Rem
```

```
110   Rem     Assign values to each element of Array
130     For Index=1 To 9 : Array(Index)=Index : Next Index
140   Rem
150   Rem     Note that Index now has a value of 10
170   Rem     Print initial values of variables.
190   @"Initial value of String$ = ";String$
200     For X=1 To 9 : @ Array(X); : Next X : @
210   @"Initial value of Index = ";Index
220   Rem
230   Rem     Call the procedure .Proc'1
250   Call .Proc'1 (String$,Mat A,Index)
260   Rem
270   Rem     Print the variables that were initialized earlier
280   Rem     and then passed to the procedure.
300   @ : @"End value of String$ = ";String$
310     For X=1 To 9 : @ Array(X); : Next X : @
320   @"End value of Index = ";Index
330   Rem
340   Rem     Note that Index still has value of 10 (not passed back).
360   End
380   Rem     Procedure will modify all values and then print them.
390   Rem     Note that Index is *not* passed back to main program.
410 Procedure .Proc'1 (X$,Mat A,Index)
420   X$="....................."
430     For X=1 To 9 : A(X)=10-X : Next X
440   Index=123456789.0
450   @ : @ Tab(10);"Value of String$ inside procedure = ";X$
460   @ Tab(10); :   For X=1 To 9 : @ A(X); : Next X : @
470   @ Tab(10);"Value of Index inside procedure = ";Index
480   Endproc
```

After you have run the first program successfully, modify lines 250 and 480 like this:

```
250 Call .Proc'1 (String$,Mat A,Index;Index)
480 Endproc (Index)
```

Run the program and note the final value of Index: it has been passed back from the procedure.

The second program demonstrates how a procedure can call itself. Notice that when the procedure returns to the procedure that called it, the value of X is the same as before the call, even though the first level executed itself 4 more times.

Example 2

```
10   Rem      Recursive procedure demonstration
20   Rem      This procedure will call itself and print
30   Rem      level or nesting going in and out.
40   Rem
50   X=0 : @"Main level"
60   Call .Recursive'procedure (X)
70   @"Main level"
80   End
100 Procedure .Recursive'procedure (Number)
110   @ Tab(Number*5);"Just entered level #";Number
120   If Number<5 Then  Do
130     Call .Recursive'procedure (Number+1)
140     Enddo
150   @ Tab(Number*5);"Now leaving level #";Number
160   Endproc
```

The third set of programs should be entered and then saved separately under the names Root, One, Two, and Three. Once they are all saved on disk, create the library **testlib** and add the programs One, Two, and Three to it, using LIBBUILD. List the library directory to be sure they are all in the library. Then run Root.

Root

```
5   Rem      Root program
10   Dim String1$(99)
20   Rem
25   Rem      Assign an initial value to String1$
30   String1$="This is message number zero "
40   Rem
45   Rem      Identify Basic library to be searched for procedures.
50   Library"Testlib"
60   @"main"
70   Rem      Call procedures to demonstrate nesting of procedures
75   Rem      and loading from a library.
80   Call .One (String1$)
90   @"main"
99   End
```

One

```
100   Rem     Library procedure one
110 Procedure .One (X$)
120   @ Tab(5);"Begin one"
130   @ Tab(5);X$
140   X$="Message from procedure one"
150   Call .Two (X$)
160   @ Tab(5);X$
170   @ Tab(5);"End one"
180   Endproc
```

Two

```
200   Rem     Library procedure two
210 Procedure .Two (X$)
220   @ Tab(10);"Begin two"
230   @ Tab(10);X$
240   X$="Message from procedure two"
250   Call .Three (X$)
260   @ Tab(10);X$
270   @ Tab(10);"End two"
280   Endproc
```

Three

```
300   Rem     Library procedure three
310 Procedure .Three (X$)
320   @ Tab(15);"Begin three"
330   @ Tab(15);X$
340   X$="Message from procedure three"
350   Endproc
```

program: **Library Builder**

format: **Run "LIBBUILD"**

The LIBBUILD program allows the user to create and modify a Procedure Library. LIBBUILD will prompt the user for the necessary file references.

Note:

1. A Library may be composed of many different Modules or just one SAVE file. The LIBBUILD Program is required to group a number of different SAVE files into a single Library which can automatically be searched for a needed Procedure. All files requested by the LIBrary BUILDer must be SAVEd Basic files. Each file (or Module) may contain one or more Procedures.

   A CALL to a Procedure which is in the Current Library will cause all Procedures in the same Module (SAVE file) to be loaded into one Partition. If only one Procedure is to be loaded into each Partition, then only one Procedure may be SAVED at a time. This will ensure that there is only one Procedure in each Module and therefore only one Procedure in each Partition.

Example:

As an example of creating and building a LIBRARY, let us assume that we have a program which consists of a main program, two Procedures (one of which usually calls the other several times in succession), and three larger Procedures (no two of which need to be in memory at the same time). The two smaller Procedures are named .AA and .BB while the larger ones are .XX, .YY, and .ZZ.

While in Basic, ENTER the two Procedures .AA and .BB into the User Area. Then issue a SAVE command. We will call this SAVE file MOD1. Then, in turn, ENTER and SAVE the three large Procedures. These SAVE files will be named MOD2, MOD3, and MOD4.

Next, LOAD and RUN the LIBBUILD program (supplied on the disk with Cromemco Structured Basic.)

The first selection from the program menu must be C for Create Procedure Library. The name of the Library may be any legal file name.

Continue to follow the prompts of LIBBUILD and Add the four SAVE files to the Library which was just created.

If the Library name is specified in a LIBRARY instruction in a Basic program, a CALL to any of the four Procedures will automatically load the Procedure into the highest numbered available Partition (refer to the CALL instruction.)

instruction: **Procedure Call**

format: **[Ln] [CALL] .prname [(parset1; parset2)]**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

.prname is a procedure name

parset1 is an optional set of one or more actual parameters to be passed to the called PROCEDURE. These parameters may include one or more of the following:

mvar an implicitly or explicitly DIMensioned one, two, or three DIMensional matrix. In this list the name of the matrix must be preceded by the word MAT.

svar a string variable.

aexp an arithmetic expression, variable, or constant.

parset2 is an optional set of one or more return parameters whose values are received from the called PROCEDURE. Parset2 is composed of arithmetic variables only.

The CALL instruction transfers control to a PROCEDURE.

Notes:

1. Parset1 is composed of variables which are to be transferred to the PROCEDURE. This list must match the list in the PROCEDURE definition (refer to the PROCEDURE instruction).

2. Parset 2 contains return variables. This list must match the list in the ENDPROC or EXITPROC instruction (refer to the ENDPROC and EXITPROC instructions).

3. The semicolon (;) is part of parset2. It must precede parset2 but is not to be included if parset2 is not present.

4. Implementation of Procedure Library search:

   When a PROCEDURE is referenced by a Basic program, the following locations will be searched until it is found:

   1. the Current Partition,

   2. all other Partitions, commencing with Partition zero.

If the PROCEDURE is located in one of the Partitions, control will be transferred to the PROCEDURE in that Partition. If 1 and 2 above are not successful, the search will continue with:

3. the Current Library.

If the PROCEDURE is found within the Library, the Module containing the PROCEDURE will be loaded into the highest numbered Available Partition. An Available Partition is one which is neither manually nor automatically locked. Manual locking is invoked by the use of the Lock instruction. Locking occurs automatically when nested calling of PROCEDUREs (in other Partitions) takes place. When control is transferred out of a Partition by a PROCEDURE CALL, that Partition is locked. When control is transferred out of a Partition by an ENDPROC, ERRPROC, or EXITPROC instruction, that Partition is unlocked (assuming that no active control structures remain).

If the search is not successful, an error will be generated.

5. The variables in parset1 are automatically declared as local to the CALLed Procedure. This is advantageous when calling another Procedure within the same Partition, or when a Procedure CALLs itself (a recursive Procedure).

Example:

```
 100        Rem Program to demonstrate the automatic
 110        Rem local feature of a Procedure CALL.
 120        Rem
 130        Dim String'one$(15),String'two$(15)
 140        String'one$="AAAAAAAAAAAAAAAA"
 150        String'two$="BBBBBBBBBBBBBBB"
 160        Print : Print String'one$ : Print String'two$
 170        Call .String'proc (String'one$)
 180        Print : Print String'one$ : Print String'two$
 190        End
 200        Rem
 210        Rem
1000       Procedure .String'proc (String'two$)
1100        Print : Print String'one$ : Print String'two$
1200        String'two$="CCCCCCCCCCCCCCC"
1300        Print : Print String'one$ : Print String'two$
1400        Endproc
1500        End

>>Run
AAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBB

AAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
```

```
CCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCC

CCCCCCCCCCCCCCCC
BBBBBBBBBBBBBBB
***190 End***
```

The user may wish to refer to the Glossary while reading the following explanation.

This example demonstrates that parameters which are received by Procedure are local to that Procedure. In the main program, the string variables String'one$ and String'two$ are DIMensioned, assigned values of 16 A's and 16 B's, and displayed for verification.

When .String'proc is CALLed, String'one$ is passed to the Procedure in which it is referred to as String'two$. This is a call by reference. This means that during the execution of the Procedure the two string variables will be equivalent to each other. The two strings are displayed again. String'one$ maintains its value of A's because all variables which are not used as receiving parameters in a Procedure are global within a Partition. String'two$ is local to the CALLed procedure because it is a receiving parameter. As such, String'two$ received the original value of String'one$ and now it also displays a value of A's.

The next instruction assigns a string of C's to String'two$. String'one$ takes on this same value because the two string variables are equivalent to each other during the execution of the Procedure. The two strings are once again displayed for verification.

Upon the termination of the Procedure (ENDPROC) and a return to the main program, the two string variables are displayed a final time. String'one$ (global throughout the Partition) maintains its latest value and contains all C's. String'two$ (which was local to the Procedure) reverts to the value it had at the time the Procedure was CALLed and contains all B's.

statement: **Procedure Definition**

format: **Ln PROCEDURE .prname**

Ln PROCEDURE .prname (parset-1)

where:

Ln is a line number

.prname is a Procedure name

parset1 is an optional set of one or more formal parameters which are passed from the calling program. These parameters may include one or more of the following:

mvar a one, two, or three DIMensional matrix which is defined by the calling routine. In this list the name of the matrix must be preceded by the word MAT.

svar a string variable.

avar an arithmetic variable.

The PROCEDURE definition names a PROCEDURE and provides an entry point into the PROCEDURE.

Note:

1. Parset 1 contains receiving variables. These variables must match parset1 in the corresponding Procedure CALL.

statement: **Procedure End**

format: **Ln ENDPROC**

Ln ENDPROC (parset2)

where:

Ln is a line number

parset2 is an optional set of one or more actual return parameters whose values are passed to the calling program. Parset2 can contain arithmetic expressions, variables, and constants.

The ENDPROC statement indicates a logical end of a PROCEDURE and returns control to the calling program.

Note:

1. Parset2 contains expressions whose values are to be returned to the calling program. These must match parset2 in the corresponding Procedure CALL.

statement: **Procedure Error End**

format: **Ln ERRPROC**

where:

Ln is a line number

The ERRPROC statement returns control to the calling program in case of a user trapped error. The statement removes all active control structures within the current PROCEDURE and sets the Basic error flag before transferring control to the calling program.

Notes:

1. The ERRPROC instruction is useful when an error is detected while one or more WHILE, UNTIL, etc. control structures are pending in a called PROCEDURE. A jump in program logic to an ERRPROC instruction will cause the Basic error flag to be set, the run time stack to be scrubbed back to the PROCEDURE CALL, and control to be transferred to the calling routine.

2. This statement sets SYS(3) = 251. SYS(3) stores the last error encountered in the Basic program.

3. No parameters may be passed by the ERRPROC statement.

statement: **Procedure Exit**

format: **Ln EXITPROC**

Ln EXITPROC (parset2)

where:

Ln is a line number.

parset2 is an optional set of one or more actual return parameters whose values are passed to the calling program. Parset2 can contain arithmetic expressions, variables, and constants.

The EXITPROC statement returns control to the calling program when it is not possible for a normal End of Procedure (ENDPROC) to be executed. The statement removes all active control structures within the current PROCEDURE and transfers control to the calling program.

Notes:

1. Parset2 contains expressions whose values are to be returned to the calling program. These must match parset2 in the corresponding PROCEDURE CALL.

2. The EXITPROC instruction is useful when an error or other need for a change in program logic is detected while one or more control structures are pending in a called PROCEDURE. The execution of an EXITPROC statement will cause the runtime stack to be scrubbed back to the last PROCEDURE CALL and control to be transferred to the calling program.

instruction: **Clear Partition**

format: **[Ln] CLEAR aexp**

[Ln] CLEAR .prname

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

aexp is an arithmetic expression, variable, or constant which represents a partition number in the range 0-7.

.prname is the name of a procedure.

The CLEAR instruction parallels the SCRatch instruction, but affects only the specified Partition, or the Partition containing the specified PROCEDURE.

Note:

1. The CLEAR instruction overrides the LOCK instruction.

instruction: **Select Procedure Library**

format: **[Ln] LIBRARY**

[Ln] LIBRARY svar

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

svar is a string variable or string literal file reference.

When used with a file reference, the LIBRARY instruction causes the specified file to become the Current LIBRARY.

If the file reference is omitted, the LIBRARY instruction closes the Current LIBRARY.

Notes:

1. The Current LIBRARY is the file which will be searched if a requested PROCEDURE is not current in one of the Partitions. Refer to the PROCEDURE CALL instruction.

2. The LIBRARY can be closed by the LIBRARY, CLOSE, SCR, or BYE instructions. If the CLOSE instruction is used, it must not specify a file number (i.e., CLOSE all files).

3. The LIBRARY instruction automatically closes the Current LIBRARY prior to opening a new LIBRARY.

4. The LIBRARY may be a single SAVE file.

command: **Use Partition**

format: **USE aexp**

USE .prname

where:

aexp is an arithmetic expression, variable, or constant which represents a partition number in the range 0-7.

.prname is the name of a procedure.

If the USE instruction specifies a Partition number, that Partition will become the Current Partition.

If a PROCEDURE name is specified, then the Partition containing that PROCEDURE will become the Current Partition. If the specified PROCEDURE is not located in one of the Partitions, the Current LIBRARY will be searched. If the PROCEDURE is found within the LIBRARY, the Module containing the procedure will be loaded into the highest numbered Available Partition and this will become the Current Partition.

Note:

1. The USE command allows the user to hand load and EDIT Modules.

instruction: **Lock Partition**

format: **[Ln] LOCK aexp**

[Ln] LOCK .prname

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

.prname is the name of a Procedure.

The LOCK instruction causes the specified Partition to become not Available for automatic Module loading. If the LOCK instruction specifies a single PROCEDURE, the entire Partition containing that Module is LOCKed.

Note:

1. The LOCK instruction overrides the automatic lock/unlock feature.

instruction: **Unlock Partition**

format: **[Ln] UNLOCK aexp**

[Ln] UNLOCK .prname

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

.prname is the name of a procedure.

The UNLOCK instruction causes the specified Partition to become Available for automatic Module loading. If the UNLOCK instruction specifies a single PROCEDURE, the entire Partition containing that Module is UNLOCKed.

Note:

1. The UNLOCK instruction does not override the automatic Lock feature.

Chapter 21

PROGRAM SECURITY

This chapter describes the instructions used to provide for program security by preventing the listing of some or all of the program.

command: **Delete Remark Statements**

format: **DELREM**

DELREM L1

DELREM L1,

DELREM L1,L2

where:

L1 is an optional line number or line name.

If L1 is omitted, all lines of the program are processed.

If L1 is the only argument and there is no comma following it, then L1 is the only line processed.

If L1 is the only argument and the comma is included, L1 through the last line in the program are processed by the command.

L2 is an optional line number or line name which indicates the last line to be processed. If included, it must be preceded by L1 and a comma.

The DELREM command DELetes all REMark statements which occur in the lines specified by L1 and L2.

Notes:

1. The DELetion of REMark statements from a program will reduce the amount of memory space needed to RUN it and disk space needed to store it. However, it will have virtually no effect on the execution time of the program.

2. Because DELeted REMark statements are not recoverable, the following procedure is recommended for the use of DELREM:

   a. After a program has been debugged, LIST or SAVE a copy of the program on a disk file before using DELREM.

   b. Use DELREM to delete REMark statements.

   c. SAVE a copy of the new, shorter version of the program in a different file. The file extension NSV could be used to indicate No remarks, SaVed file.

command: **Protect Program Lines**

format: **NOLIST**

NOLIST L1

NOLIST L1,

NOLIST L1,L2

where:

L1 is an optional line number or line name.

If L1 is omitted, all lines of the program are protected.

If L1 is the only argument and there is no comma following it, then L1 is the only line protected.

If L1 is the only argument and the comma is included, L1 through the last line in the program are protected by the command.

L2 is an optional line number or line name which indicates the last line to be protected. If included, it must be preceded by L1 and a comma.

The NOLIST command allows the programmer the option of keeping part or all of a Basic file confidential. After the execution of the NOLIST command, the specified lines cannot be LISTed.

Notes:

1. Because NOLISTed lines can never be LISTed, the following procedure is recommended for the use of NOLIST:

   a. After a program has been debugged, Save or LIST a copy of the program in a disk file before using NOLIST.

   b. Use NOLIST to protect those parts of the program which are confidential.

   c. SAVE a copy of the NOLISTed version of the program in a different file. One could use the file name extension PUB to indicate that it is a PUBlic file.

2. This command protects only those lines present at the time the command is executed. If, after the NOLIST command has been executed new lines are entered, they will not be protected from LISTing even if they replace already NOLISTed lines.

Chapter 22

BASIC-KSAM

Basic-KSAM is the **K**eyed **S**equential **A**ccess **M**ethod file management system, which is a part of Cromemco 32K Structured Basic.

It was developed primarily for use in applications where large files are involved and fast random access is necessary. Such applications include, but are not limited to, inventory control, reservation systems, library systems, accounts receivable, and bill of materials processing.

Files are always accessed dynamically, i.e., sequential and random access instructions can be intermixed freely. Records are identified by a unique user defined data field within the record called the Primary Key. Additionally any number of fields may be designated as Alternate Keys (sometimes called Secondary Keys) for retrieval purposes. Examples of keys are: part numbers for inventory control, account numbers for billing systems, and customer names for mailing list applications.

Records can be accessed randomly by their Primary Key or by any of their Alternate Keys.

In addition, Basic-KSAM supports sequential movement through the file (forward, backward and static), and random access by partial key or relative record number.

Space is automatically allocated to the file when records are added. Any number of files can be processed simultaneously provided that sufficient memory for buffer storage is available.

If a diskette becomes full during processing, the user program may instruct Basic-KSAM to overflow automatically to additional disks without program interruption.

Basic-KSAM utilizes a look-aside buffering technique which eliminates unnecessary disk accesses thereby increasing retrieval speed and minimizing head movement, as well as drive and media wear.

A number of utility programs (KSAMUT) and Numeric Sorting Conversion functions (IKEY$, FKEY$, KEYI, and KEYF) are also provided as part of Basic-KSAM.

LOGICAL STRUCTURE

Basic-KSAM files consist of three portions:

a. The data set
b. The key set
c. The header

Data Set

The data set contains the actual data records created by the user. These records are arranged in ascending collating sequence on the basis of their Primary Keys. As records are added or deleted, Basic-KSAM keeps the entire file in this sequence.

Records are organized in Blocks (Data Blocks) which are scattered throughout the available disk space. The number of records per Data Block is derived by dividing the block length by the record length. Logical records may not span Data Blocks so that only an integral number of records may be contained in any one Block. Any remaining bytes are left unused.

Basic-KSAM allocates space to a file in multiples of a page (256 bytes). Each Data Block normally occupies four pages (1K bytes).

Key Set

The key set contains various pointers to the Data Blocks and it is maintained by Basic-KSAM.

Header

The header contains information about the structure and contents of the file such as record length, key length, and key displacement. It is created from information supplied by the user at file-creation time and maintained by Basic-KSAM. It occupies 1K bytes.

PHYSICAL STRUCTURE

A Basic-KSAM file is organized physically as a collection of one or more disk files. These files may reside on one or more volumes (disks). Each file has its own name which is specified at file creation time. The purpose of having more than one file name associated with a Basic-KSAM file is the creation of a multi-volume file.

A Basic-KSAM file may extend to more than one disk. The user specifies the file names to be included in the Basic-KSAM file at file opening, and can increment this number at any time during processing by use of the KADDVOL instruction.

All volumes of a given Basic-KSAM file must be mounted and on-line any time the file is accessed.

Logical Records and Keys

Logical records make up the largest portion of a Basic-KSAM file. Records are added to Data Files by the user, and to Alternate Key files by Basic-KSAM at the request of the user.

All records of a given Data File must be of the same length. Data File record length is user defined and may vary from one byte to one Data Block.

Alternate Key file record length is determined by the criteria outlined in the following section.

Records may not span Data Blocks. This implies that the minimum number of records in a Block is one. A Data Block may not span two volumes but Blocks of the same file may be distributed over several volumes (diskettes). The maximum number of records in a Block is derived by dividing the Block length (default value of 1024 bytes) by the record length, and taking the highest integer which is less than or equal to the quotient.

Unused Block Space

When more than one record can fit in a Data Block, the user may specify at file-creation time that a portion of the block be reserved for later addition of records. Although this may waste space on the disk it improves processing speed greatly in those applications where frequent additions to the file are involved.

Keys

Any field within the record can be used as a means of randomly accessing the record. A field used for that purpose is called a Key. Key fields may be up to 250 bytes in length.

All Keys are stored by Basic-KSAM as strings. If the user needs a numeric variable (Integer, Long or Short Floating Point) as a Key, the Basic-KSAM Numeric Sorting Conversion Instructions (IKEY$, FKEY$, KEYI and KEYF) must be used. These will convert numbers to strings which may be sorted by Basic-KSAM and will also convert these strings back into numeric variables which can be used by Basic. Refer to Chapter 18 for a description of these instructions.

Primary Key

One of the record keys must contain a value which is unique for that record and is used to identify and distinguish it from all other records of the same file. This key is called the Primary Key and is declared as such by the user when creating the file. The length and displacement of the Primary Key must be the same for all records of the same file.

Alternate Keys

All other keys are considered to be Alternate Keys and need not be unique. The length and displacement of an Alternate Key must also be the same for all records of the same file.

Examples of Primary Keys are account numbers, part numbers, and Social Security numbers. Examples of Alternate Keys are zip codes and surnames.

Alternate Keys are used to create pointer (index) files called Alternate Key or Inversion Files. These files provide a means of accessing records by the contents of a field other than the Primary Key.

Key Length

The Primary Key may be from one to 250 bytes long. If Alternate Keys are also used, the sum of the longest Alternate Key plus the Primary Key may not exceed 250 bytes.

Record Retrieval

For random retrieval the user normally specifies the key of the desired record when requesting service from Basic-KSAM.

For example if a customer record must be retrieved and the Primary Key consists of the customer account number, that number must be supplied if random access is desired. Of course, if sequential retrieval is acceptable the file can be read sequentially (forward or backward) until the desired record is found.

The relative position of a record within the file may also be used for retrieval. This is a number ranging from 0 (for the first record) to the actual number of records in the file minus 1. Once a record has been read in this fashion, it can be updated or deleted as desired.

The user accessing records by relative position must note that the relative position of records will change as records are added or deleted.

ALTERNATE KEY FILES

An Alternate Key is a field in the record which may be used as an alternate means of retrieving the record.

Any and all fields in the record may be designated as Alternate Keys at any time after the file is created. The difference between the Primary Key and any Alternate Key is that Alternate Key values need not be unique. For example if a file of students exists, the Primary Key may be the student's social security number while the major field of study may be used as an Alternate Key. Since more than one student may have the same major, the Alternate Key value will not be unique but the social security number (Primary Key) will be. Student

records may then be accessed in order of their major rather than their social security number, thus eliminating the need for sorting the file in major sequence.

An Alternate Key file (inversion file) is a Basic-KSAM file whose records are arranged in ascending order by the value of the Alternate Key field and contain pointers to corresponding records in the Primary or Data File. Any number of Alternate Files may be created for the same Data file, each corresponding to a different Alternate Key field of the Data file record. An Alternate Key file record is an inversion of the corresponding data file record according to the specified Alternate Key. Since Alternate Key files are Basic-KSAM files they can also be read independently of their associated Data file for inquiry and report purposes. The user cannot, however, add, delete, or update records in the Alternate Key file independently using the KADD, KPUT, KDEL, and KUPDATE instructions. To add or delete records from an Alternate Key file the appropriate Alternate Key instructions must be used while both the Data and Alternate Key files are open.

There is no restriction that there be a one-to-one correspondence between the Data file records and their corresponding records in the Alternate Key file. Adding or deleting Alternate Key file records is not done automatically upon adding or deleting a Data file record, but is left up to the user. One may therefore create Alternate Key records for only a part of the total Data file. Similarly a Data file record may be deleted or altered without its corresponding Alternate Key record being affected. If the user wishes to maintain a one-to-one correspondence then a record must be added to every Alternate Key file after a new Data file record is added, and deleted before the Data file record is deleted. If a Data file record is to be modified so that the value of an Alternate Key field will be changed, and the user wishes to maintain a one-to-one correspondence, the affected Alternate Key file record must be deleted first, then added again after the Data file record is updated. This will insure that the latest value of the Alternate Key field is stored in the Alternate Key file and that the Data file record can still be accessed through the Alternate Key. If this is not done, as a result of the selective addition and deletion of Alternate Key file records, it is possible that these records may be left stranded in the sense that they are still part of the Alternate Key file but they cannot be used to access any Primary Data file records. This should be avoided as it defeats the purpose of the Alternate Key file.

Alternate Key file records are made up of two fields. The first field contains the value of the Alternate Key field in the corresponding Primary Data file record. Therefore the length of this field equals the length of the corresponding Alternate Key field in the Data file record. The second field contains the Primary Key of the corresponding Data file record and is equal in length to the Primary Key.

The total length of the Alternate Key File record is the sum of the lengths of the Alternate Key and Primary Key fields.

In the example above, if the Social Security field (Primary Key) of the Data file record is 9 bytes and the major code (Alternate Key) is 2 bytes, each record in the Alternate Key file MAJOR will be 11 bytes long.

Since Alternate Key files are Basic-KSAM files, they must also possess their unique Primary Key. Therefore the entire Alternate Key File record constitutes the Primary (and only) Key of the Alternate Key file. Since the Primary Key of the Data file record is unique, the Primary Key of the Alternate Key file is also unique.

Taking these facts into consideration the user is never required to supply the record length, Primary Key length and Primary Key displacement when creating an Alternate Key file. All that is needed is the length and displacement in the Data file record of the field to be used as an Alternate Key. The file number of an open Data file is passed to Basic-KSAM and these quantities are calculated as follows:

```
RECORD LENGTH OF ALTERNATE KEY FILE =
LENGTH OF ALTERNATE KEY FIELD +
LENGTH OF PRIMARY KEY OF DATA FILE RECORD

PRIMARY KEY LENGTH OF ALTERNATE KEY =
RECORD LENGTH OF ALTERNATE KEY

PRIMARY KEY DISPLACEMENT OF ALTERNATE KEY FILE = 0
```

Quantitative inquiries about the Data file can be made without ever accessing it. If the user keeps a one-to-one correspondence between the Data file and all of its Alternate Key files, questions such as:

How many students are enrolled in Engineering?

can be answered quickly without having to access the Data file. After the Data file has been opened, the Alternate Key file MAJOR is opened, and all records corresponding to major = Engineering are counted. Since the Alternate Key File records are generally much smaller than the corresponding Date file records, Alternate Key files can be scanned very quickly.

Temporary Alternate Key files can be created instead of sorting for report generation purposes. These files can be deleted immediately after the desired report is produced. With a little effort the user can implement inquiry programs to quickly answer questions such as:

How many AMERICAN MALE students are enrolled in ENGINEERING?

The words in capitals correspond to fields in the Data file record which can be used as Alternate Keys.

Remembering that the Alternate Key file record contains the Primary Key of the Data file record corresponding to it, the Data file can always be accessed directly (without the use of Alternate Key instructions) by using information derived from Alternate Key Files.

Note that Basic-KSAM requires that the Primary File be opened in order to open the Alternate Key File. The Primary file need not be accessed.

THE CURRENT RECORD POINTER (CRP)

At all times Basic-KSAM maintains a pointer to the current record. Special CRP cases are the beginning of file (BOF) and end of file (EOF). All possible positions of the CRP are described in detail below:

1. Any successful read, add or update operation will cause the CRP (current record pointer) to point to the successfully processed record.

2. A BOF (beginning of file) condition occurs whenever the CRP points to the dummy record in front of the first logical record of the file. This is the case immediately after the file is opened. Reading backwards (sequentially) past the first record will also set this condition.

3. When the EOF (end of file) is reached the CRP (current record pointer) points to the dummy record beyond the last record of the file. This condition will occur when reading sequentially forward past the last record, or when a random instruction is given with a key value greater than any key on file.

4. A successful deletion will cause the CRP to point to the previous record (in the collating sequence of the Primary Keys) or to the BOF if the first record of the file was deleted.

5. An unsuccessful read, update or delete operation will position the CRP at the first record which has a key value greater than the one specified in the operation. The CRP will be left pointing to the EOF if the specified key is larger than any on file or if sequential reading past the last logical record was attempted.

6. An unsuccessful add operation will position the CRP at a record in the file which has the same key as the one specified by the user. (Remember that Primary Keys must be unique so that trying to add a record with a key that already exists will cause an error.) An exception to this rule is the case of an add which was unsuccessful because there was no more room available on the disk. In this instance the CRP points as specified in paragraph E.

7. If the CRP is positioned at the BOF and a read previous or read current operation is attempted, the CRP is unchanged. This is also true if the CRP is positioned at EOF and a read current or read next is attempted.

8. If an I/O error occurs during processing the contents of the CRP is unreliable.

   When in doubt as to the position of the CRP, issue a Read Current Record, Primary File (KGETCUR) or retrieve key (KRETRIEVE) instruction. This will return either the current record or a BOF/EOF/File Empty error message.

SAMPLE BASIC-KSAM FILE HANDLING PROGRAM

This is a Structured Basic Program which demonstrates some Basic-KSAM file building and accessing techniques.

```
 200   Rem Program to demonstrate the establishment and use
 300   Rem of Basic-KSAM Primary Data and Alternate Key files.
 400   Rem
 500   Rem
 600   Dim Supplier$(6),Part$(0),Class$(0)
 700   Rem create and open Primary Data File
 800   Rem Primary File Layout:
 900   Rem        Primary Key- Part Number
1000   Rem          bytes  1-7 Supplier Name
1100   Rem                 8   Shipping Class
1200   Kcreate\8,1\"Primary"
1300   Kopen\1\"Primary"
1400   Rem
1500   Rem create and open two Alternate Key Files
1600   Kaltcreate\1,7\"Supplier"
1700   Kaltopen\2,1\"Supplier"
1800   Kaltcreate\1,1,7\"Class"
1900   Kaltopen\3,1\"Class"
2000   Rem
2100   Rem
2200   Read Part$,Supplier$,Class$
2300     While Part$#"0"
2400     Rem build Primary Data file
2500     Kadd\1,Part$\Supplier$,Class$
2600     Rem put corresponding records in two Alternate
2700     Rem Key files
2800     Kaltadd\2\
2900     Kaltadd\3\
3000     Read Part$,Supplier$,Class$
3100     Endwhile
3200   Rem close files, data base is complete
3300   Kclose\1\ :  Kclose\2\ :  Kclose\3\
3400   Rem
3500   Rem
3600   Rem Open files for query
3700   Kopen\1\"Primary"
3800   Kaltopen\3,1\"Class"
3900   Rem error indicates that there are no more
4000   Rem records with the specified Alternate Key
4100   On Error Goto Done
4200   Rem
4300   Rem get first occurrence in Primary Data file of
4400   Rem record with specified Alternate Key
4500   Kaltfirst\3,"1"\Supplier$,Class$
4600   Rem retrieve Primary Key
4700   Kretrieve\1\Part$
4800   Print Part$,Supplier$,Class$
4900   Rem loop through successive records with specified
```

```
 5000   Rem Alternate Key
 5100  *Loop :  Kaltfwd\3\Supplier$,Class$
 5200   Kretrieve\1\Part$
 5300   Print Part$,Supplier$,Class$
 5400   Goto Loop
 5500  *Done :  Kclose\1\ :  Kclose\3\
 5600   Rem erase files so the program may be run again
 5700   Erase"Primary" :  Erase"Supplier" :  Erase"Class"
 5800   End
 5900   Rem
10000   Data"9","Fred T.","1"
10001   Data"4","Mary Q.","3"
10002   Data"3","Fred T.","2"
10003   Data"7","Jane S.","1"
10004   Data"2","Jane S.","3"
10005   Data"1","Fred T.","2"
10006   Data"0","0","0"

>>Run
7                    Jane S.              1
9                    Fred T.              1
***5800 END***

>>
```

Line 600 DIMensions string variables. A variable which is DIMensioned as 0 contains one element (numbered 0).

Lines 1200 and 1300 create and open the Primary Data file (named PRIMARY) with a record length of 8 and a Key length of 1. Channel number 1 is used for this file.

Lines 1600-1900 create and open two Alternate Key files. The first (SUPPLIER) relates the supplier name (Alternate Key) to the part number (Primary Key). SUPPLIER uses the file which was opened on channel 1 (PRIMARY) as the Primary Key file and has a Key length of 7 with a displacement (by default) of 0.

Note that the displacement does <u>not</u> include the Primary Key. The second Alternate Key file (CLASS) relates the shipping class (Alternate Key) to the part number (Primary Key). CLASS also uses the file which was opened on channel 1 (PRIMARY) as the Primary Key file and has a Key length of 1 with a displacement of 7.

Lines 2200-3100 READ the data from the DATA statements and build the Primary Data and two Alternate Key files. Notice that in line 2500 the Primary Key is not part of the variable list.

Line 4500 searches the Alternate Key file CLASS for the first record containing a shipping class of 1. If one is found, the Primary Data file (PRIMARY) is searched for the corresponding record. The class and supplier name are then read from the Primary File. Notice that the Primary Key has not been returned. The next instruction (line 4700) retrieves the current Primary Key.

The KALTFWD instruction searches for additional records in the Alternate Key file which have the same Alternate Key value.

When no more records with the same Alternate Key value exist, an error is generated, trapped by line 4100, and control is passed to line 5500 (line label Done).

The techniques used in this example can be drawn upon and extended to provide the user with examples of the use of all of the Basic-KSAM instructions. As will become self evident, the power of these instructions will allow the user to accomplish many things which were heretofore impossible in Basic.

BASIC-KSAM INSTRUCTIONS

Basic-KSAM instructions are logically divided into five groups:

A. Instructions which treat the entire file as a unit.

1. Create Primary Data File
2. Create Alternate Key File
3. Close File
4. Open Primary File
5. Open Alternate File
6. Add Volume to Existing File

B. Sequential access in Primary Key sequence.

1. Read Previous Record, Primary File
2. Read Next Record, Primary File

C. Random access by Primary Key.

1. Read Random Record, Primary File
2. Read Approximate, Primary File
3. Update Record, Primary File
4. Delete Record, Primary File
5. Read Nth Record, Primary File
6. Add Record, Primary File
7. Load Record, Primary File

D. Current record access

1. Read Current Record, Primary File
2. Retrieve Primary Key, Current Record
3. Read Fields, Current Record
4. Write Fields, Current Record

E. Alternate Key instructions.

1. Read Primary Record by Current Alternate Key
2. Read First Primary Record by Specified Alternate Key
3. Read Next Primary Record by Current Alternate Key
4. Verify Alternate Record
5. Add Record, Alternate File
6. Delete Record, Alternate File

Instructions in group A, with the exception of the Close instruction, expect the file to be closed. If it is not, an Invalid Request error will occur. The Close instruction expects the file to be open.

As was mentioned earlier, it is possible that at a certain point in time the file may be devoid of data records. This will happen when the only record on the file is deleted or when the file has just been created. If an Empty File is closed the only acceptable instructions are KOPEN and KALTOPEN. If the Empty File is open the only acceptable instructions are LOAD, ADD, and CLOSE. All other requests will return an EMPTY-FILE error.

When the file becomes empty it does not disappear. The file remains open until closed by the user (KCLOSE).

Since I/O errors are possible with every instruction they will not be explicitly mentioned where error return codes are discussed. This is also true for all system related errors such as File Not Found, etc. Note that the Invalid Request code can be returned by any instruction.

Refer also to Chapter 18, Machine Level Instructions, KSAM Numeric Sorting Conversions. These instructions are necessary to create key fields from numeric variables.

Summary of Instructions

INSTRUCTION	DESCRIPTION
Add Record, Primary File	Add a record to a Data file in the proper sequence by Primary Key.
Add Record, Alternate File	Add a record to an Alternate Key file corresponding to the current Primary Data file record.
Add Volume to Existing File	Add a volume to an existing file.
Close File	Close a file.

Create Primary Data File	Initialize and format a Basic-KSAM Primary Data file.
Create Alternate Key File	Initialize and format a Basic-KSAM Alternate Key file.
Delete Record, Primary File	Delete a record from a Primary Data file.
Delete Record, Alternate File	Delete the Alternate Key file record (if any) corresponding to the current Primary Data file record.
Load Record, Primary File	Load (add) a record to a Primary Data file and attempt to preserve the unused space in the block. Usually used when building the file.
Open Alternate File	Open an Alternate Key file for further processing.
Open Primary File	Open a Primary Data file for further processing.
Read Previous Record, Primary File	Read the previous Primary File record in the collating sequence of the Primary Keys.
Read Current Record, Primary File	Read the current record from the Primary Data file.
Read Fields, Current Record	Read data from the current record from the Primary Data file.
Read Next Record, Primary File	Read the next Primary file record in the collating sequence of the Primary Keys.
Read Random Record, Primary File	Read a record randomly from the Primary Data file by Primary Key.
Read Approximate, Primary File	Read the first record in the Primary Data file whose Primary Key is greater than or equal to a given key.
Read Nth Record, Primary File	Read the Nth record of the Primary Data file.

Read Primary Record By Current Alternate Key	Read the Primary Data file record corresponding to the current Alternate record.
Read First Primary Record By Specified Alternate Key	Read the first Primary Data file record which contains a given Alternate Key value.
Read Next Primary Record, Current Alternate Key	Read the next Primary Data file record which contains the same Alternate Key value as the current Alternate Key file record.
Retrieve Primary Key, Current Record	Read the Primary Key of the current record.
Update Record, Primary File	Replace a given record in a Primary Data file.
Verify Alternate Record	Read the Alternate Key File record (if any) corresponding to the current Primary Data File record. Check that the contents of the Alternate record correspond to the Primary record.
Write Fields, Current Record	Write data to the current record in the Primary Data File.

File Instructions

Before a file can be accessed it must be created. This is the function of the KCREATE (KALTCREATE) instruction. This instruction does not actually add any data records to the file (this is done through the KADD (KALTADD), KPUT, and KLOAD instruction), it only pre-formats certain fixed areas and allocates initial disk space to the file.

After the file is created it exists in an empty state until a record is written to it. Note that a file may be created in one program but actually be built in another. After the KCREATE (KALTCREATE) instruction is processed successfully, the file is automatically closed by Basic-KSAM.

Before an existing file can be processed, it must be opened through the KOPEN (KALTOPEN) instruction. An Invalid Request will be returned if any request is issued to an unopened file.

After processing of a file is completed, the file must be closed by the KCLOSE instruction. It is very important to remember to close a file if records were added, deleted or updated.

Refer also to the Alternate Key Instructions section.

instruction: **Create Primary Data File**

format: **[Ln] KCREATE\pr1,pk1\file-ref-1,file-ref-2,...**

[Ln] KCREATE\pr1,pk1,spc\file-ref-1,file-ref-2,...

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

pr1 — Primary Data file record length (not including Primary Key)

pk1 — Primary Key Length (maximum of 250 bytes)

spc — unused space per block

file-ref — is one or more file references to the Primary Data File. This is a Cromix path name composed of optional directory names and a file name separated by slashes. The file references may be string variables or string literals (enclosed in quotation marks.)

The KCREATE instruction will initialize and format one or more Basic-KSAM files.

Notes:

1. Errors which may be returned:

Error	Meaning
167	Invalid Request
168	Invalid Create Parameters
169	File Exists

Error Result: The file is not created.

2. If more than one file reference is used, the Basic-KSAM file may span more than one volume (disk). More than one file reference must be used for multi-volume files.

3. An attempt will be made by the KLOAD instruction to preserve the unused space per block. This instruction can be used for building a Data file, while the Kadd instruction is generally used to add records after the initial file set up.

4. Refer also to the Create Alternate Key File (KALTCREATE) instruction.

5. A null (empty) string used as a file reference indicates to Basic that all subsequent file references in the list are to be ignored.

```
10   Vol1$ = "/a/File1"
20   Vol2$ = "/b/File2"
30   Vol3$ = ""
40   Kcreate \50,7\ Vol1$,Vol2$,Vol3$,Vol4$
```

Because Vol3$ is a null string a 2 volume file (Vol1$ and Vol2$) is created.

instruction: **Close File**

format: **[Ln] KCLOSE\fn**

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

fn — Primary Data or Alternate Key file number

The KCLOSE instruction will close and update the status of a Basic-KSAM file. This instruction is valid for both Primary Data and Alternate Key files.

Notes:

1. Errors which may be returned:

Data File	Alternate File	Meaning
167	183	Invalid Request

Error Result: The file is not closed.

2. This request is normally the last one issued after completion of file processing. Not issuing this instruction will result in data being lost if records were added to or deleted from the file, or if any records were updated. If the file is open it must be closed before a Rename or Erase operation is performed. If this instruction is issued while the file is closed then an Invalid Request code will be returned.

3. The CLOSE instruction will CLOSE all open files (including Basic-KSAM files).

instruction: **Open Primary File**

format: **[Ln] KOPEN\pfn\file-ref-1,file-ref-2,...**

where:

[Ln] is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

pfn Primary Data file number

file-ref is one or more file references to the Primary File. This is a Cromix pathname composed of optional directory names and a file name separated by slashes. The file references may be string variables or string literals (enclosed in quotation marks.)

The KOPEN instruction makes an existing file available for further processing.

Notes:

1. Errors which may be returned:

Error	Meaning
167	Invalid Request
173	Open Failed

Error Result: The file is not opened.

2. The file must exist (it must have been previously created).

3. The number of file references in the KOPEN instruction must be the same as the number of file references in the corresponding KCREATE instruction. More than one file reference must be used for multi-volume files.

4. A null (empty) string used as a file reference indicates to Basic that all subsequent file references in the list are to be ignored.

```
10   Vol1$ = "/a/File1"
20   Vol2$ = "/b/File2"
30   Vol3$ = ""
40   Kopen 1 Vol1$,Vol2$,Vol3$,Vol4$
```

Because Vol3$ is a null string a 2 volume file (Vol1$ and Vol2$) is opened.

5. Each open Basic-KSAM file uses 512 + 256 * (Sys(11) + Sys(12)) bytes. The values for Sys(11) and Sys(12) are determined at the time the file is created. Using the default values of 4 pages per Key Block and 4 pages per Data Block, each Basic-KSAM file will occupy 2304 decimal bytes.

instruction: **Add Volume To Existing File**

format: **[Ln] KADDVOL\fn\file-ref**

where:

[Ln] is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

fn is a Primary Data or Alternate Key file number.

file-ref is a single file reference to the file which is to be added to the Primary or Alternate file. This is a Cromix path name composed of optional directory names and a file name separated by slashes. The file reference may be a string variable or string literal (enclosed in quotation marks.)

The KADDVOL instruction allows the user to add an additional volume (disk) to an already created file. This is necessary when the file is in an overflow condition.

Note:

1. This instruction is valid for both Primary Data and Alternate Key files.

Sequential Access Instructions

These instructions do not require that a key be supplied by the user. Instead, they use the Current Record Pointer (CRP) which was discussed previously.

The Read Previous (KGETBACK) instruction will decrement the CRP unless it is pointing to the BOF (Beginning of File) in which case it will be left unchanged.

The Read Next (KGETFWD) instruction will increment the CRP unless it is pointing to the EOF (End of File) in which case it will leave it unchanged.

Although none of these instructions return the Primary Key, it may be obtained by executing a KRETRIEVE instruction following any other instruction.

instruction: **Read Previous Record, Primary File**

format: **[Ln] KGETBACK\pfn**

[Ln] KGETBACK\pfn\var-1,...,var-n

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
pfn	Primary Data file number
var1-n	numeric and/or string variable list

The KGETBACK instruction returns the previous record in Primary Key sequence.

Notes:

1. Errors which may be returned:

Error	Meaning
161	File Empty
162	Beginning of File
167	Invalid Request

Error Result: No movement of data takes place

2. If the CRP is pointing to the EOF, it is decremented and the last record of the file is returned.

If the CRP is pointing to the first record of the file it is decremented so that it points to the BOF, a Beginning of File error is returned, no data is moved, and the CRP is left pointing to the BOF.

If the CRP is pointing to the BOF, a Beginning of File error is returned.

In all other cases the CRP is decremented and the record to which it is pointing (next lower Primary Key on file) is returned.

instruction: **Read Next Record, Primary File**

format: **[Ln] KGETFWD\pfn**

[Ln] KGETFWD\pfn\var-1,...,var-n

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
pfn	Primary Data file number
var1-n	numeric and/or string variable list

The KGETFWD instruction returns the next record in Primary Key sequence.

Notes:

1. Errors which may be returned:

Error	Meaning
161	File Empty
163	End of File
167	Invalid Request

Error Result: No movement of data takes place

2. If the CRP is pointing to the BOF, it is incremented and the first record of the file is returned.

   If the CRP is pointing to the last record of the file an End of File error will be returned, no movement of data will take place, and the CRP will be incremented so that it points to the EOF.

   If the CRP points to the EOF an End of File error will be returned, no movement of data will take place, and the CRP will be left pointing to the EOF.

   In all other cases the CRP is incremented and the record to which it is pointing (the next higher Primary Key on file) is returned.

Random Access Instructions

These instructions use the key (or record number) rather than the CRP to locate the desired record. On completion, successful or otherwise, the CRP is adjusted according to the rules in the section describing the Current Record Pointer (CRP). This allows sequential processing to follow any random instruction.

The following discussion applies to all random access instructions except the Read Nth Record (KGETREC) instruction:

Basic-KSAM will attempt to locate a record in the Primary Data file whose key matches the specified key.

a. If such a record exists, KADD or KLOAD operations will fail with an Invalid Key return code (Primary Key must be unique) while all other operations will be successful. The CRP will point to that record. If a Delete instruction is successfully completed the CRP will then be moved back by one record (previous record in the sequence of Primary Keys) since the original record will no longer be there.

b. If the given key is higher than any key in the Primary Data file, KADD or KLOAD operations will append this record to the end of the file and set the CRP pointing to it. All other operations will fail with an End of File return code and set the CRP pointing to EOF.

c. If the given key does not exist, and it is less than the highest key on file, the KADD and KLOAD operations will add this record in the correct place according to the sequence of the Primary Keys, and the Read Approximate (KGETAPP) instruction will retrieve the next higher key on file. The CRP will be positioned at that record. All other operations will fail with an Invalid Key return code. The CRP will be pointing to the next higher key on file.

The Read Nth Record (KGETREC) instruction will attempt to locate the record by its sequence number (in the order of Primary Keys) rather than its key. If found, the CRP will point to it. If not, an End of File return code is set and the CRP points to the EOF.

instruction: **Read Random Record, Primary File**

format: **[Ln] KGETKEY\pfn,pkey\var-1,...,var-n**

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
pfn	Primary Data file number
pkey	Primary Key (string variable or literal)
var1-n	numeric and/or string variable list

The KGETKEY instruction will return the record with the specified Primary Key.

Notes:

1. Errors which may be returned:

Error	Meaning
161	File Empty
163	End of file
164	Invalid Key
167	Invalid Request

Error Result: No movement of data takes place

2. If a record exists on file with the given key the operation is successful. The record is moved to the variable list and the CRP is set pointing to it.

If the given key is higher than the highest key on file the operation fails with an End of File return code and the CRP points to the EOF.

If the given key is less than the highest key on file but it is nonexistent the operation fails with an Invalid Key return code. The variable list is not altered and the CRP points to the next record (next higher key).

instruction: **Read Approximate, Primary File**

format: **[Ln] KGETAPP\pfn,pkey\var-1,...,var-n**

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
pfn	Primary Data file number
pkey	Primary Key (string variable or literal)
var1-n	numeric and/or string variable list

The KGETAPP instruction will return the record with the specified Primary Key or the next higher Primary Key.

Notes:

1. Errors which may be returned:

Error	Meaning
161	File Empty
163	End of file
167	Invalid Request

   Error Result: No movement of data takes place

2. If a record exists on file with the given key or a higher key the operation is successful. The record is moved to the variable list and the CRP is set pointing to it.

   If the given key is higher than the highest key on file the operation fails with an End of File return code and the CRP points to the EOF.

3. This is a very useful tool for browsing through a file. The advantage it has over read random is that it will normally fail only if the given key is higher than the highest on file. This means that the exact key need not be known for successful retrieval. A partial key with the lower portion filled with blanks will either retrieve the desired record or the record with the next higher key. Sequential processing normally follows this operation. For example if a group of records exists for which the high order portion of the key is the same, a KGETAPP instruction will retrieve the first of the group and the rest can be accessed sequentially. If the key supplied is higher than any on file and the End of File error occurs, the CRP is positioned at the EOF. Otherwise a record is retrieved and the CRP points to it.

4. The exact Primary Key for the returned record may be obtained by executing a KRETRIEVE instruction following the KGETAPP instruction.

instruction: **Update Record, Primary File**

format: **[Ln] KUPDATE\pfn,pkey**

[Ln] KUPDATE\pfn,pkey\var-1,...,var-n

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
pfn	Primary Data file number
pkey	Primary Key (string variable or literal)
var1-n	numeric and/or string variable list

The KUPDATE instruction will update the record with the specified Primary Key.

Notes:

1. Errors which may be returned:

Error	Meaning
161	File Empty
163	End of file
164	Invalid Key
167	Invalid Request

Error Result: No movement of data takes place

2. This operation behaves in the same manner as a read random operation. The difference between the two instructions is the direction of the data movement.

instruction: **Delete Record, Primary File**

format: **[Ln] KDEL\pfn,pkey**

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
pfn	Primary Data file number
pkey	Primary Key (string variable or literal)

The KDEL instruction will delete the record with the specified Primary Key.

Notes:

1. Errors which may be returned:

Error	Meaning
161	File Empty
163	End of file
164	Invalid Key
167	Invalid Request

Error Result: No record is deleted

2. If the record does not exist in the Primary Data file, this operation will fail in the same manner as the Read Random instruction, the return codes and CRP settings are the same. If the record exists, it will be physically removed from the file and the space occupied by it will be reclaimed. The CRP will be set pointing to the previous record (next lower key) or to the BOF if the first record on file is deleted. The return code will indicate successful completion. If the only record of the file is deleted, then the file exists in an Empty state until a record is written to it.

instruction: **Read Nth Record, Primary File**

format: **[Ln] KGETREC\pfn,rec**

[Ln] KGETREC\pfn,rec\var-1,...,var-n

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
pfn	Primary Data file number
rec	record number
var1-n	numeric and/or string variable list

The KGETREC instruction will return the record whose number in the sequence of Primary Keys is rec.

Notes:

1. Errors which may be returned:

Error	Meaning
161	File Empty
163	End of File
167	Invalid Request

Error Result: No movement of data takes place

2. For successful retrieval the record number (rec) must be in the range from 0 to the actual number of records in the file minus one. The first record in the file is considered to be record 0 (relative position). If the number is within range the operation is successful. The record is returned and the CRP points to it. If not, the operation fails with an End of File return code and the CRP points to the EOF.

instruction: **Add Record, Primary File**

format: **[Ln] KADD\pfn,pkey\var-1,...,var-n**

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
pfn	Primary Data file number
pkey	Primary Key (string variable or literal)
var1-n	numeric and/or string variable list

The KADD instruction will add the record (variable list) to the Primary Data file in sequence by the Primary Key.

Notes:

1. Errors which may be returned:

Error	Meaning
164	Invalid Key
167	Invalid Request
170	Key Set Full
171	No Free Blocks

Error Result: No movement of data takes place

2. The record is inserted between two others which carry the next lower and next higher keys or at the EOF (BOF), if its key is higher (lower) than any key in the file. The CRP points to the record.

If a record with the same key already exists, the operation fails with Invalid Key code and the file is not altered. The CRP is left pointing to that record (which can be retrieved by the Read Current Record (KGETCUR) instruction).

If a free block cannot be found on any of the disks allocated to the file, the operation will fail with a No Free Blocks return code and the contents of the file will not be altered. The CRP will be left pointing either to the next higher key or to the EOF. A Retrieve Primary Key (KRETRIEVE) instruction will establish the position of the CRP.

The file should either be closed and compacted (see UTILITIES), or the number of volumes should be incremented (KADDVOL). Then the KADD instruction may be issued again.

The KADD instruction will disregard the unused space per block option of the KCREATE instruction, even if specified by the user, and will attempt to fit as many records in a block as the block length permits. To preserve unused space the user should write records to the file using the KLOAD instruction.

instruction: **Load Record, Primary File**

format: **[Ln] KLOAD\pfn,pkey\var-1,...,var-n**

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
pfn	Primary Data file number
pkey	Primary Key (string variable or literal)
var1-n	numeric and/or string variable list

The KLOAD instruction will add the record (variable list) to the Primary File in sequence by the Primary Key.

Notes:

1. Errors which may be returned:

Error	Meaning
164	Invalid Key
167	Invalid Request
170	Key Set Full
171	No Free Blocks

   Error Result: No movement of data takes place

2. This instruction has practical use only when adding successively higher keys to the end of the file. This might be the case, for example, when building the file from scratch using records which are in the same sequence as the file being created. In this instance the unused space, specified when the KCREATE instruction was issued, will be preserved. This is the only difference between the KADD and KLOAD instructions. If the records are in different sequence or if this instruction is used to insert records anywhere other than the end of the file, KLOAD behaves in the same manner as the KADD instruction (i.e., it attempts to fill the blocks completely).

Current Record Instructions

These instructions read or write from whatever record the Current Record Pointer (CRP) is pointing to. They do not change the position of the CRP in any way. After the execution of one of these instructions, the CRP continues to point to whatever record it pointed to before the instruction executed.

instruction: **Read Current Record, Primary File**

format: **[Ln] KGETCUR\pfn**

[Ln] KGETCUR\pfn\var-1,...,var-n

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
pfn	Primary Data file number
var1-n	numeric and/or string variable list

The KGETCUR instruction returns the current record from the Primary Data file.

Notes:

1. Errors which may be returned:

Error	Meaning
161	File Empty
162	Beginning of File
163	End of File
167	Invalid Request

Error Result: No movement of data takes place

2. The CRP is not changed. If it is pointing to the BOF or EOF then the appropriate error code is returned and no movement of data takes place. In all other cases the record to which the CRP is pointing is moved to the variable list. This instruction is normally used in conjunction with random operations.

instruction: **Retrieve Primary Key, Current Record**

format: **[Ln] KRETRIEVE\pfn\svar**

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
pfn	Primary Data file number
svar	string variable

The KRETRIEVE instruction returns the Primary Key from the current record in the Primary Data file.

Notes:

1. Errors which may be returned:

Error	Meaning
161	File Empty
162	Beginning of File
163	End of File
167	Invalid Request

Error Result: No movement of data takes place

2. This instruction is similar to the Read Current instruction. The difference is that only the Primary Key of the current record is returned upon successful completion.

3. The KRETRIEVE instruction does not alter the CRP. It returns the Primary Key from the record to which the CRP is pointing.

instruction: **Read Fields, Current Record**

format: **[Ln] KGET\pfn\var-1,...,var-n**

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

pfn — Primary Data file number

var1-n — numeric and/or string variable list

The KGET instruction returns the named variables from the current record in the Primary Data file.

Notes:

1. Errors which may be returned:

Error	Meaning
161	File Empty
162	Beginning of File
163	End of File
167	Invalid Request

Error Result: No movement of data takes place

2. The Current Record Pointer (CRP) must be positioned before the KGET instruction is used. KGET does not alter the CRP. It only returns data from the record to which the CRP is pointing. If the CRP is pointing to the BOF or EOF, then the appropriate error code is returned and no movement of data takes place.

For example, the two following sets of instructions are equivalent:

```
100  Kgetfwd\1\Id'number, Year, Amount
```

and

```
100  Kgetfwd\1\
110  Kget\1\Id'number, Year
120  Kget\1\Amount
```

instruction: **Write Fields, Current Record**

format: **[Ln] KPUT\pfn\exp-1,...,exp-n**

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
pfn	Primary Data file number
exp1-n	numeric and/or string variable list

The KPUT instruction writes the expressions to the current record in the Primary Data file.

Notes:

1. Errors which may be returned:

Error	Meaning
167	Invalid Request
174	I/O Error

Error Result: No movement of data takes place

2. The Current Record Pointer (CRP) must be positioned before the KPUT instruction is used. KPUT does not alter the CRP. It only returns data from the record to which the CRP is pointing. If the CRP is pointing to the BOF or EOF, then the appropriate error code is returned and no movement of data takes place.

Alternate Key Instructions

Alternate Key instructions allow the user to create and maintain Alternate Key (inversion) files and access the Primary Data file by means of Alternate Keys.

All Alternate Key instructions involve two files, the Primary Data file and one of its Alternate Key files.

The Primary Data File must be open when any Alternate Key instruction is issued and, with the exception of KALTCREATE and KALTOPEN, the Alternate Key file must also be open.

During the execution of an Alternate Key instruction an error can occur while accessing either the Primary Data file or the Alternate Key file. If an error occurs on the Primary Data file record, then the return code will range from 161 to 174. If an error occurs on the Alternate Key file, then the return code will range from 177 to 190.

As was mentioned earlier, all Basic-KSAM instructions, with the exception of KCREATE, KADD, KLOAD, and KUPDATE, can be issued to an Alternate Key file. In this case Basic-KSAM treats the file as a Primary Data file and if any error occurs then returned error codes range from 161 to 174.

The KALTCUR, KALTFIRST, and KALTFWD instructions do <u>not</u> return the Primary Key. The Primary Key can be obtained after the execution of one of these instructions by executing a KRETRIEVE instruction.

instruction: **Create Alternate Key File**

format: **[Ln] KALTCREATE\pfn,akl\file-ref-1,file-ref-2,...**

[Ln] KALTCREATE\pfn,akl,akd\file-ref-1,...

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

pfn — Primary Data file number

akl — Alternate Key length not including the Primary Key (maximum of 250 bytes when combined with Primary Key.)

akd — Alternate Key displacement not including the Primary Key. Default value of 0.

file-ref — is one or more file references to the Alternate File. This is a Cromix path name composed of optional directory names and a file name separated by slashes. The file references may be string variables or string literals (enclosed in quotation marks.)

The KALTCREATE instruction will create and format one or more Basic-KSAM Alternate Key Files.

Notes:

1. Errors which may be returned:

Error	Meaning
183	Invalid Request
184	Invalid Create Parameters
185	File Exists

Error Result: The file is not created.

2. If more than one file reference is used, the Basic-KSAM file may span more than one volume (disk). More than one file reference must be used for multi-volume files.

3. The Alternate Key displacement does not include the Primary Key. At this point the Primary Key is transparent to the user and need not be taken into consideration.

4. A null (empty) string used as a file reference indicates to Basic that all subsequent file references in the list are to be ignored.

```
10    Vol1$ = "A:File1"
20    Vol2$ = "B:File2"
30    Vol3$ = ""
40    Kopen \1\ Vol1$,Vol2$,Vol3$,Vol4$
```

Because Vol3$ is a null string a 2 volume file (Vol1$ and Vol2$) are opened.

instruction: **Open Alternate File**

format: **[Ln] KALTOPEN\afn,pfn\file-ref-1,file-ref-2,...**

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

afn — Alternate Key file number

pfn — Primary Data file number

file-ref — is one or more file references to the Alternate File. This is a Cromix path name composed of optional directory names and a file name separated by slashes. The file references may be string variables or string literals (enclosed in quotation marks.)

The KALTOPEN instruction makes an existing Alternate Key file available for further processing.

Notes:

1. Errors which may be returned.

Error	Meaning
183	Invalid Request
189	Open Failed

Error Result: The file is not opened.

2. The file must exist (it must have been previously created).

3. More than one file reference must be used for multi-volume files.

4. A null (empty) string used as a file reference indicates to Basic that all subsequent file references in the list are to be ignored.

```
10   Vol1$ = "A:File1"
20   Vol2$ = "B:File2"
30   Vol3$ = ""
40   Kopen \1\ Vol1$,Vol2$,Vol3$,Vol4$
```

Because Vol3$ is a null string a 2 volume file (Vol1$ and Vol2$) are opened.

instruction: **Read Primary Record By Current Alternate Key**

format: **[Ln] KALTCUR\afn\var-1,...,var-n**

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

afn — Alternate Key file number.

var1-n — numeric and/or string variable list

The KALTCUR instruction will read the Primary Data file record specified by the current record in the Alternate Key file.

Notes:

1. Errors which may be returned:

Data File	Alternate File	Meaning
161	177	File Empty
163	179	End of File
164	-	Invalid Key
-	178	Beginning of File
167	183	Invalid Request

Error Result:

No movement of data takes place. The CRP for the Primary Data file follows the rules of the KGETKEY instruction. The CRP of the Alternate Key file remains unchanged.

2. An Invalid Key error code can be returned by this instruction. This error will occur when:

   a. There is no record in the Primary Data file whose Primary Key matches the Primary Key field on the current record of the Alternate Key file (stranded inversion due to deletion of the Primary file record).

   b. A record exists in the Primary Data file with the corresponding Primary Key but its Alternate Key value does not match the value of the alternate field on the current record of the Alternate Key file (stranded inversion due to modification of the Alternate Key field in the Primary Data file record).

instruction: **Read First Primary Record By Specified Alternate Key**

format: **[Ln] KALTFIRST\afn,akey**

[Ln] KALTFIRST\afn,akey\var-1,...,var-n

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
afn	Alternate Key file number
akey	Alternate Key (string variable or literal)
var1-n	numeric and/or string variable list

The KALTFIRST instruction will read the first (in Primary Data file sequence) Primary file record specified by the Alternate Key.

Notes:

1. Errors which may be returned:

Data File	Alternate File	Meaning
161	177	File Empty
163	179	End of File
164	-	Invalid Key
167	183	Invalid Request

Error Result:

No movement of data takes place. The CRP for the Primary Data file follows the rules of the KGETKEY instruction. The CRP for the Alternate Key file follows the rules of the KGETAPP instruction.

instruction: **Read Next Primary Record By Current Alternate Key**

format: **[Ln] KALTFWD\afn**

[Ln] KALTFWD\afn\var-1,...,var-n

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

afn — Alternate Key file number.

var1-n — numeric and/or string variable list

The KALTFWD instruction will read the next (in Primary Data file sequence) Primary Data file record specified by the current record in the Alternate Key file.

Notes:

1. Errors which may be returned:

Data File	Alternate File	Meaning
161	177	File Empty
163	179	End of File
167	183	Invalid Request

Error Result:

No movement of data takes place. The CRP for the Primary Data file follows the rules of the KGETKEY instruction. The CRP for the Alternate Key file follows the rules of the KGETFWD instruction.

2. This instruction will attempt to locate the next Primary Data file record that contains the same Alternate Key field value as the current Alternate Key record. Assume that we have issued a KALTFIRST instruction on an Alternate Key file. Successive executions of KALTFWD instruction will access all logical records of the Primary Data file with the same Alternate Key value.

instruction: **Verify Alternate Record**

format: **[Ln] KALTVER\afn**

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

afn Alternate Key file number.

The KALTVER instruction attempts to locate the Alternate Key file record which corresponds to the current Primary Data file record.

Notes:

1. Errors which may be returned:

Data File	Alternate File	Meaning
161	177	File Empty
163	179	End of File
162	-	Beginning of File
-	180	Invalid Key
167	183	Invalid Request

Error Result:

If a corresponding record does not exist on the given Alternate Key file an Invalid Key error will be returned. The CRP for the Data File follows the rules of the KGETKEY instruction. The CRP of the Alternate file follows the rules of the KGETFWD instruction.

instruction: **Add Record, Alternate File**

format: **[Ln] KALTADD\afn**

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

afn — Alternate Key file number

The KALTADD instruction will write a record in the specified Alternate Key file. This record will correspond to the current record of the Primary Data file.

Notes:

1. Errors which may be returned:

Data File	Alternate File	Meaning
161	-	File Empty
163	-	End of File
162	-	Beginning of File
167	183	Invalid Request
-	186	Key Set Full
-	187	No Free Blocks

Error Result:

The alternate record is not added. The CRP for the Primary Data file is not changed. The CRP for the Alternate Key file follows the rules of the KADD instruction.

instruction: **Delete Record, Alternate File**

format: **[Ln] KALTDEL\afn**

where:

Ln — is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

afn — Alternate Key file number.

The KALTDEL instruction will delete a record from the specified Alternate Key file. This record is the one which corresponds to the current record of the Primary Data file.

Notes:

1. Errors which may be returned:

Data File	Alternate File	Meaning
161	177	File Empty
163	-	End of File
162	-	Beginning of File
-	180	Invalid Key (No Alternate Key record to Delete)
167	183	Invalid Request

Error Result:

The record is not deleted. The CRP for the Primary Data file is not changed. The CRP of the Alternate Key file follows the rules of the KDEL instruction.

KSAMUT UTILITY PROGRAM

The Ksamut utility program performs a number of functions useful for creating and maintaining KSAM files. The functions are:

COMMAND	DESCRIPTION
CHANGE DISKS	Change the disks and initialize the bit maps.
CREATE	Create a Primary Data or Alternate Key file.
ERASE	Remove a Basic-KSAM file from the directory
COMPACT	Compact the file
REORGANIZE	Reorganize the file
STATUS	Print file status
RENAME	Rename a file
COPY COMPACT	Copy the file in compacted form
COPY REORGANIZE	Copy the file in reorganized form

To load the utility program, type KSAMUT in response to the operating system prompt. The utility then displays a list of functions on the console screen. The user can select one function at a time. If the selected command needs more information before processing, then the user must follow the prompts displayed on the screen by the utility program. If a file has to be accessed then the file unit, name, and type of the first volume must be provided. If the file extends to more volumes you will be asked to provide file unit, name, type information for all volumes. All volumes of the file must be mounted.

Any number of requests can be executed on the same or different files of disks without exiting from the utility.

When the utility is executed, all Basic-KSAM and operating system error codes are possible. If the execution of any request is successful the utility responds with "SUCCESSFUL", displays the menu on the screen again, and waits for the next request. If the request failed then the appropriate message is printed and the utility returns to the menu waiting for the next request.

In this discussion, file-ref indicates a file reference of the form:

[X:]FILENAME.XYZ

In the above file reference, X is an optional disk drive specifier. If omitted, the file is assumed to be in the current directory. "B:" through "H:" are mapped to "/b" through "/h" on the Cromix Operating System.

Note that all commands and file references given while running KSAMUT must be given in upper case characters.

KSAMUT Prompts

The KSAMUT commands use several prompts to request additional information from the user. This section describes the prompts. Note that this list describes every prompt used by all the commands. Each individual command uses a different subset of the prompts.

Remember that all responses to the prompts must be given in upper case.

Prompt	**ENTER FILE UNIT**
Response	Enter the disk drive designator or directory name where the file resides.
Prompt	**ENTER FILE NAME**
Response	Enter the file name without giving the file name extension
Prompt	**ENTER FILE TYPE**
Response	Enter the file name extension. Do not precede the extension with a period (.).
Prompt	**NUMBER OF VOLUMES**
Response	Enter the number of volumes that the file may extend to. The number must be >=1 and <=4.
Prompt	**PAGES PER KEY BLOCK**
Response	Enter the number of pages per key block. The number of keys in a key block must be 2<n<256. To determine the maximum number of pages for any given key length (K), use the formula **Pages=INT(((K+5)*255)/256).** For alternative files, the key length is the alternate key length plus the primary key length.

Prompt **PAGES PER DATA BLOCK**

Response Enter the number of pages per data block. The best size for the data block is the multiple of 256 that divided by the record length plus key length yields the smallest remainder. For example, with a record length of 163 and a key length of 2, the number of pages would be 11. This is arrived at by dividing 165 (163+2) into successive multiples of 256. Using 11*256=2816, 2816/165 gives a remainder of 11. This means that only 11 bytes of every 2816 bytes will be wasted.

Note: This method optimizes disk space usage, not processing speed.

Prompt **FREE SPACE**

Response Enter the amount of free space in terms of logical records to be reserved in each block when the file is built using the Kload instruction. The amount of free space should be less than the maximum number of records in a block.

Specifying free space is useful only when the file is built with the Kload instruction and the Kadd and Kput instructions are used to add records to the file. Specifying unused space can optimize file processing speed when many additions will be made to the file. If the file will be static or many deletions are expected, the amount of free space should be zero.

Prompt **RECORD LENGTH**

Response Enter the length of the record in bytes.

Prompt **KEY LENGTH**

Response Enter the length of the primary key in bytes.

Prompt **KEY DISPLACEMENT**

Response Enter the position of the first byte of the key within the record.

KSAMUT Commands

The following paragraphs describe each Ksamut command in more detail:

Request **CHANGE DISKS**
Definition Change the disks and initialize the bit maps.

Discussion — This command must be executed if the user wishes to change disks while running KSAMUT. After the completion of this command, Drive A (or the current directory) is the current disk drive.

Request Definition — **CREATE**
Create a Basic-KSAM Primary Data or Alternate Key file.

Discussion — This command will create a Basic-KSAM file. The user must follow the prompts and provide the needed information. Also refer to the KCREATE and KALTCREATE instructions.

Request Definition — **ERASE**
Remove a Basic-KSAM file from the disk directory.

Discussion — If the file extends to more than one disk then all disks should be mounted. If the operation was successful then, all disk space occupied by this file has become available for other use. The file name has been removed from all the disk directories to which it extended. If the operation failed then the appropriate message is printed.

Request Definition — **COMPACT**
Rewrite the original file so that its blocks contain the maximum number of logical records.

Discussion — If the execution of this request is successful, then the new file has blocks containing the maximum number of logical records. The file name(s) remain the same. The new file might have fewer data or key blocks in use, but the disk space occupied by the compacted file will be the same as the original.

If the execution of this request fails then the appropriate error message is printed.

Request Definition — **REORGANIZE**
Rewrite the original file so that every block contains the initial unused space specified at file-creation time.

Discussion — This command tries to rewrite the given file so that blocks contain if possible the unused space specified at file creation time. The disk space occupied by the new file will be the same as the original file.

A compact file will not be reorganized.

NOTE — The COPY COMPACT and COPY REORGANIZE commands are more efficient than the COMPACT and REORGANIZE commands.

Request Definition **STATUS**
Print file status (Header Information)

Discussion The following will be printed:

FILE NAME
FILE TYPE
RECORD LENGTH
INITIAL UNUSED SPACE
PAGES PER KEY BLOCK
PAGES PER DATA BLOCK
DATA BASE (OR ALTERNATE FILE)
KEY LENGTH
KEY DISPLACEMENT
VOLUMES SPECIFIED
VOLUMES IN USE
DATA BLOCKS IN USE
FILE SIZE IN RECORDS

Request Definition **RENAME**
Rename a file

Discussion This command is similar to the rename command of the operating system. The file to be renamed does not have to be Basic-KSAM file. This command does not need to have all volumes of a Basic-KSAM file mounted to be executed.

Request Definition **COPY and COMPACT**
Copy the given file so that the blocks of the new file contain the maximum number of records.

Discussion The successful execution of this request will create a compact copy of the original file. The new file can be created on the same or different disks. The new file does not have to extend to the same number of volumes as the original file. If the new file will extend to disks on which the original file exists then their filenames must be different.

The number of data blocks, key blocks and the disk space occupied by the new file could be less than the original file. If the operation fails, the appropriate message is printed.

Request Definition

COPY and REORGANIZE
Copy the original file so that the blocks of the new file contain the initial unused space specified file-creation time.

Discussion

This request is similar to COPY-COMPACT request. Instead of compacting the blocks, however, it attempts to insert in each block the amount of empty space specified by the user at file creation time.

The disk space occupied by the new file might be more than that of the original file.

The COPY-COMPACT and COPY-REORGANIZE commands are the only commands by which the free data and key blocks are physically removed from the file.

ERROR CODES FOR KSAM FUNCTIONS

This section describes the error codes that can be generated by Basic-KSAM functions. These error codes are also briefly described in Chapter 24.

The following table summarizes the error codes.

ERROR	PRIMARY DATA FILE	ALTERNATE KEY FILE
FILE EMPTY	161	177
BEGINNING OF FILE	162	178
END OF FILE	163	179
INVALID KEY	164	180
INVALID REQUEST	167	183
INVALID CREATE PARAMETERS	168	184
FILE EXISTS	169	185
KEY SET FULL	170	186
NO FREE BLOCKS	171	187
OPEN FAILED	173	189
I/O ERROR	174	190

Error **FILE EMPTY**
Error Codes 161 For A Primary Data File
177 For An Alternate Key File

Discussion This error code is returned when an attempt was made to access a record of an Empty file. A file is Empty if no records have been written to it, or if all its records have been deleted.

Provided that an Empty file is open, the only allowable requests are KCLOSE, KLOAD and ADD. If an empty file is closed the only request allowed is KOPEN, RENAME, and ERASE.

Remarks Upon return from a Create (KCREATE) or Create Alternate (KALTCREATE) request the file is empty and closed. Before records can be written to the file, it must be opened.

Assume that we have deleted all the records of a file. This file is still open and empty. If we want to ERASE it we must close it first.

Error **BEGINNING OF FILE (BOF)**
Error Codes 162 For A Primary Data File.
178 For An Alternate Key File.

Discussion An attempt was made to read a record before the first record of the file. This error will be returned by a Read Previous (KGETBACK), Read Current (KGETCUR), or Retrieve (KRETRIEVE) instruction.

Assume that the current record pointer (CRP) is pointing to the first record of a file. A Read Previous instruction will result in a BOF error condition. If an Open instruction is followed by Read Current or Read Previous instruction a BOF error will result.

Error **END OF FILE (EOF)**
Error Codes 163 For A Primary Data File
179 For An Alternate Key File.

Discussion This error code will be returned when an attempt is made to access a record beyond the last record of the file. The Read Next (KGETFWD), Read Current (KGETCUR), Read Random (KGETKEY), Delete (KDEL), Read Approximate (KGETAPP), Retrieve (KRETRIEVE), Update (KUPDATE) instructions as well as most Alternate Key instructions can return an EOF error message.

Error **INVALID KEY**
Error Codes 164 For A Primary Data File
180 For An Alternate Key File

Discussion This error code can be returned by random access and Alternate Key instructions.

When writing a record to a file this error indicates that a record with the same Primary Key already exists.

When reading, updating or deleting a record the error indicates that there is no such record on file.

Error **INVALID REQUEST**
Error Codes 167 For A Primary Data File
183 For An Alternate Key File

Discussion This error code can be returned by all Basic-KSAM instructions.

An Invalid Request error will be returned if:

a. A KOPEN instruction is given while the file is open.

b. A KCLOSE, sequential, or random instruction is given while the file is closed.

c. An improper instruction was issued to an Alternate Key file.

d. An Alternate Key instruction was issued to a Data file.

e. The Primary Data file and the Alternate Key file were not open when an Alternate Key instruction was given.

Error **INVALID CREATE PARAMETERS**
Error Codes 168 For A Primary Data File
184 For An Alternate Key File.

Discussion The Create (KCREATE) and Create Alternate (KALTCREATE) instructions are the only ones that can return this error code. It means that the given create parameters are not correct.

An attempt is being made to create a file with a key larger than the logical record size, key length plus key displacement larger than the record length, key length equal to zero, logical record length larger than Data Block length, or with unused space per block greater or equal to the maximum number of records per block.

Error **FILE EXISTS**
Error Codes 169 For A Primary Data File
185 For An Alternate Key File

Discussion This error code can be returned by a Create (KCREATE) or Create Alternate (KALTCREATE) instruction. It means that a file with the given file name and file type already exists on the given drive.

Error **KEY SET FULL**
Error Codes 170 For A Primary Data File
186 For An Alternate Key File

Discussion This error can be returned only by a KADD, KLOAD, or Add Alternate (KALTADD) instruction. The maximum number of key blocks is 16. This error will occur because an attempt is made to allocate a key block to a file that already has 16 key blocks. The record causing the problem is not added to the file.

To continue, close the file, compact it using the utility program, and proceed by adding the last record again.

Error **NO FREE BLOCKS**
Error Codes 171 For A Primary Data File
187 For An Alternate File

Discussion This error can be returned only by the Add (KADD), Load (KLOAD), or Add Alternate (KALTADD) instruction. An attempt was made to allocate a new Data Block to the file. No disk space was found on any disk to which the file can extend.

The record which was to be written is not added to the file. If the number of volumes cannot be increased, the file must be closed and then compacted using the utility program.

Error **OPEN FAILED**
Error Codes 173 For A Primary Data File
189 For An Alternate Key File

Discussion This error is returned when the operating system is not able to open the file as requested. This can be caused by a full directory. It can also occur because the file already exists.

Error **I/O ERROR**
Error Codes 174 For A Primary Data File
190 For An Alternate Key File

Discussion The operating system returned an error code when an attempt was made to access the disk.

Chapter 23

GLOSSARY

The Glossary defines the computer terms and expressions used throughout the manual.

[ ]

Square brackets are used to indicate an optional quantity. The item enclosed in square brackets may be used, in the position indicated, at the user's discretion.

Argument

An argument is an independent variable, constant, or expression used with a Basic instruction whose value can be specified by the user to instruct Basic to perform a certain task. For example, in the instruction:

```
Print A, 3, C+7
```

A, 3, and C+7 are arguments to the Basic instruction PRINT.

ASCII

This acronym stands for American Standards Code for Information Interchange. It is an industry standard used to assign numerical codes (0 through 127) to 128 characters used as letters, numbers, arithmetic operators, various symbols, and control characters. The ASC(X) function will return the ASCII equivalent of any argument. A table of ASCII codes is provided for reference in Appendix B.

Available Partition

An Available Partition is one which is not either manually nor automatically locked. Manual locking is invoked by the use of the LOCK instruction. Locking occurs automatically when nested CALLing of PROCEDUREs (in other Partitions) takes place. When control is transferred out of a Partition by a Procedure CALL, that Partition is locked. When control is transferred out of a Partition by an ENDPROC, EXITPROC, or ERRPROC instruction, that Partition is unlocked

(assuming that no active control structures remain). The LOCK instruction overrides the automatic lock feature.

Basic Library Editor

An editor which allows the user to create (give a name to) a Library or add, delete, or replace Modules within an existing Library.

Basic Word

A Basic word, commonly called an instruction, is an alphanumeric set of characters which briefly describes the operation to be performed by the computer. Some examples of Basic words are:

List
Print
On Error
Len
Stop
Rnd

Binary Code

Binary code is defined as a code where every code element is either a 0 or a 1. Computer instructions and data for most microcomputers consist of unique, 8 bit binary codes.

Command

A command in Basic is an instruction to the computer which specifies an operation to be performed. In contrast to a Basic statement (see the Statement definition), commands are executed immediately. Commands are used primarily to manipulate or execute a program once the program has been entered. Commands have no line number preceding them.

A powerful feature of Cromemco Structured Basic is the ability to use most commands as statements. As such, they may be given line numbers and included in the body of the program for execution while the program is running.

Control Character

A control character is a non-printing ASCII character which is (usually) used to transmit control signals between a peripheral device and the computer. For example, a CONTROL-P entered from the console will cause the system printer to echo all information which is displayed on the console.

Control Structure

Control structures are (sets of) instructions which change the order of execution from the sequential line number order. In structured programming preferred structures are conditional loops and branches, which allow program flow to continue linearly, conditionally repeating or skipping over sections of code.

Current Library

The Current Library is the file which will be searched if a requested PROCEDURE is not current in one of the Partitions. Refer to the LIBRARY instruction.

Current Partition

That Partition in which execution or Editing is taking place. The contents of the Current Partition will be displayed by the LIST instruction. Refer also to the USE instruction.

Current Program

The current program is any program with which the user is currently interacting. When Structured Basic is entered, no program is current. Should the user enter text to create a new program, this program becomes the current program. If the user calls a SAVEd program from system memory, that program becomes the current program. If the user EDITs a program, it remains the current program in its edited form.

Data

The term is used in two ways. Strictly speaking, any information contained within memory or control logic is binary data. Whether this data becomes alphanumeric characters or control information depends upon the program in use.

In the other sense, data is used to refer to numerical or string information. In Basic, this numerical or string information is listed in a file or DATA statement.

Default

With certain Basic instructions, an argument may be optionally added to control a certain function. If no argument is given, the instruction defaults or reverts to a value already programmed into the Basic interpreter. For example, the default values of the arguments for the command:

Renumber

are 10,10 in Cromemco Basic. This default value for RENUMBER will produce automatic line renumbering starting with line 10 and numbering consecutive lines by increments of 10, (e.g., 10, 20, 30, 40...). To change this default value, the Basic word must be followed by an argument. For example, the command:

```
Renumber 5,5
```

will provide automatic line RENUMBERing starting with line 5 and continuing by increments of 5 (e.g., 5, 10, 15,...).

Disk Storage

A disk is a computer memory device which is used to store information. Disks are typically used in place of main memory when large amounts of information must be stored. A floppy diskette is similar in appearance to a phonograph record. Most microcomputer systems currently offer disk storage capabilities through either large or mini floppy disks. The floppy and mini floppy terms refer to the two different sizes (8 inch and 5 inch respectively) of the flexible plastic disks used with the disk assemblies.

In addition, Cromemco offers a hard fixed disk with a very large (10 megabyte) storage capacity.

Entry Point

The starting line of a PROCEDURE within a Module. The Entry Point allows Basic to locate a PROCEDURE within a Module or Library.

Expression

An expression is defined as any combination of variables, constants and operators which is evaluated as a single value or logical condition. For instance, in the statement:

```
10 Let Alpha = (Num1*Num2) + (Alpha*Num3)
```

the (Num1*Num2) + (Alpha*Num3) operation, the value of which is assigned to variable Alpha, is interpreted as an expression. In the statement:

```
10 If Alpha = Beta Then Goto 250
```

the logical comparison Alpha = Beta is called an expression and is evaluated to True (=1) or False (=0).

File or Data File

A File defines a group of related information. This information is addressed by means of a File Reference and usually resides on a floppy diskette.

File Name

This is a Cromix path name composed of optional directory names and a file name separated by slashes (/). File names may also be a file name extension (usually 3 characters), separated from the name by a dot (.).

Firmware

Firmware is the middle ground between hardware and software. This term is generally applied to specific software instructions that have been burned in or programmed into Read Only Memory (ROM).

Floating Point Mode

Floating point mode refers to a method of computer calculation in which the computer keeps track of the decimal point in each number. In Structured Basic, three formats are used to define variables: Integer, Long Floating Point, and Short Floating Point. In the Long Floating Point mode, numerical values are allowed up to 14 digits. In the Short Floating Point mode, numerical values are limited to 6 digits. The default value in Cromemco Basic is the Long Floating Point (LFP) mode.

Hardware

In comparison to firmware and software, hardware represents the actual material (or hard) elements of a computer system. Items such as circuit boards, printers, terminals, and the computer itself are considered to be hardware.

Integer

An integer is defined as a whole number, positive or negative. The following numbers are examples of integers and non-integers:

INTEGERS	NON-INTEGERS
3	3.14159
10	.66666
-5	2/3

Integer Mode

Integer mode is a format used to define variables in which one or all variables within a given program are set to integer values only.

Interactive

An interactive device is one used to achieve direct person to computer communication, and vice versa. The teletype and CRT terminals are the best known examples of interactive terminals, although many variations are possible.

I/O (Input/Output)

The I/O initials stand for Input and Output. I/O is the transfer of data between the computer system and an external device. Devices such as CRT (Cathode Ray Tube) terminals, TTY (teletypewriter) terminals, and disk drives are examples of devices that accept the input data from the user, another peripheral device, or from the computer memory, and that output data to the computer or user.

Library

A collection of one or more Basic Modules which has been put into the required Basic Library format by the Basic Library Editor.

Line Number

All lines in Basic begin with a line or statement number. For example:

```
10   Print Peaches,Pears
```

includes the statement number 10. Line numbers can be assigned manually or through the AUTOL command and may be any integer from 1 through 99999. All Basic lines have a unique number which may be used to access lines which require modification or deletion from the program.

Line Name

A Line Name follows the Line Number and may be used to access the line for EDITing, or to transfer control to the named line.

Matrix

A matrix is an array of numeric variables in a prescribed form. For example, the array:

```
 3  2  0
 1  4  6
-3  4  5
```

is a matrix with three rows and three columns. A matrix with m rows and n columns is written:

```
a11     a12     a13     ...a1n
a21     a22     a23     ...a2n
 .
 .
 .
am1     am2     am3     ...amn
```

The individual entries in the matrix are called elements or cells. For example the quantity aij in the above matrix is the element in row i and column j. Subscripts used to indicate elements always denote the row first and the column second. Cromemco Basic permits the user to define one, two, or three dimensional matrices. A two (i.e., Mij) or three (i.e., Mijk) dimensional matrix is commonly called a table. A one dimensional matrix, a matrix with n columns but only one row, is commonly called a list. For example, the matrix:

```
3, -1, 5, -8
```

is a list (or a matrix) with one row and four columns.

Memory

The computer memory is used to store information, including programs and data, for future use. Microcomputers typically use semiconductor memories, of which the two most common types are random-access memory (RAM) and read-only memory (ROM). From a hardware perspective, memory consists of an array of bistable, individually addressable elements each of which represents a single binary digit. Information can be stored either in main memory, which commonly consists of RAM or ROM, or external storage devices, which include disks, magnetic tape, and magnetic drums.

Module

One or more PROCEDUREs which have been saved under one file name using the Basic SAVE command.

Partition

A subdivision of memory while running under Basic. Memory is divided into eight Partitions, numbered zero through seven (0-7). Each Partition may be loaded with one Module.

Peripheral Device

Peripheral devices are units which are used in conjunction with a computer but which are external to the computer. Peripherals refer to devices such as printers, plotters, terminals, disk storage devices, etc., which can be connected to the computer. The computer is assumed to be the central unit and peripherals are merely support devices.

Procedure

A section of Basic code as delimited by the PROCEDURE and corresponding terminal ENDPROC, EXITPROC, or ERRPROC instructions.

Program

A computer program is a set of instructions arranged into statement lines. The instructions are used to instruct the computer to perform specified operations in a certain order. Programs are designed and written to solve a wide range of problems and are used in applications as varied as process control, data reduction, telephone systems, mathematical analysis, games, and stock market transactions.

PROM

This acronym stands for Programmable Read Only Memory. PROMs consist of an array of memory cells that can be fixed in certain patterns by the application of higher than normal voltages. These memories are said to be non-volatile; that is, when power is withdrawn the programmed pattern remains.

Recently, EPROMs, or Erasable Proms, have appeared and have found industry wide usage. EPROMs may be erased by exposure to ultraviolet light, and then re-programmed. The Cromemco Bytesaver II is designed to program EPROMs.

Protocol

Protocol is a set of conventions on the format and content of messages to be exchanged between two logical devices. Most often, differences in timing account for failure of devices to communicate. For example, a certain signal might, of necessity, be present to enable an I/O request to a microprocessor's protocol. To match a computer to a terminal, one must know the mutual handshake protocol.

RAM

RAM stands for Random Access Memory, or read-write memory. In contrast to PROMs, read-write memory can be changed as well as being read. Some RAMs (known as dynamic) retain data for only a fraction of a second and must be refreshed constantly to retain data. All RAM is volatile and must have power applied to retain data patterns.

ROM

A ROM is a Read Only Memory device that is used for storing fixed information. This information is burned in, or programmed, at specific locations when the ROM is manufactured. A ROM cannot be written into during operation. Any ROM that can later be altered is a Programmable Read Only Memory (see PROM). ROM family memories, once burned, retain their data regardless of power contingencies.

Sector

A Sector is a subdivision of a track. Generally, sectors are 128 bytes or 512 bytes in size.

Software

Software is a term used to refer to the programs, languages and procedures used in a computer. For example, the Structured Basic interpreter as well as any Basic programs are identified as software.

Statement or Statement Line

A statement in Basic is an instruction or series of instructions to the computer. A statement is defined as one line in a Basic program which is preceded by a line number. For example:

```
100    Quantity = Number'per'box * Boxes
```

is defined as a statement. Typically, a statement can contain a maximum of 132 characters.

A powerful feature of Cromemco Structured Basic is the ability to use most statements as commands. As such, they may be used without line numbers and executed immediately. This is very useful for debugging programs.

Cromemco Structured Basic also allows more than one instruction on a single statement line as long as adjacent instructions are separated by a colon (:).

String Literal

A string literal (or string) is a sequence of alphanumeric characters, spaces, and special characters. In Structured Basic, string literals must be enclosed within quotation marks. Examples of valid string literals include:

```
"Cromemco Structured Basic"

"12345"

"This program prints square roots"
```

The statement:

```
100 Print "Cromemco Structured Basic"
```

will output the string

```
Cromemco Structured Basic
```

String Variable

A string variable is a variable which may assume the value of a string literal.

Track

A Track is a physically defined circular path which is concentric with the hole in the center of a disk. It is defined by its distance from the center of the disk. With the read/write head of the disk drive located on a given track, data may be read from or written to that track. A large floppy disk has 77 tracks per side, while a small disk has 40 per side.

User Area

The User Area is the Basic workspace in which a program can be written, EDITed, and RUN. The LIST command displays the contents of the User Area.

Variable

A variable is a quantity that can assume any one of a given set of values. In Structured Basic, variables are defined by a letter (A through Z) followed by any combination of up to 30 letters, numbers, and apostrophes ('). Examples of legal variable names include:

```
Oranges

Boxes'of'Oranges

Cost'per'box

A1
```

Variables represent numeric values. In the statement:

```
20    Portion = 8 + 2
```

Portion is the variable and 8+2 or 10 is the value assigned to Portion. A new value can be assigned to Portion at a subsequent point in the program.

Chapter 24

BASIC ERROR MESSAGES

This chapter describes the error messages produced by Basic.

FATAL ERRORS

This section describes fatal errors; that is, those errors that cause the execution of a command or program to cease.

FATAL ERRORS

-----Error-----

Number	Message	Meaning
1	Syntax	This error message covers a number of errors which can occur when the user is entering (typing in) a program. For example: Unmatched parentheses: A=(B*(C) Misspelled words: Print A Wrong data type: A$=3*A Bad punctuation: Print A(7;2) Because there is only one message for all these errors, a dollar sign is printed under the line in error at a position approximately indicating the position of the error.
2	Using Syntax	The format string for a Print Using instruction is in error.

		For example: Print Using "#.##!!!", 3.2E9 (only 3 exclamation marks; 4 required)
3	Number of Arguments	A function call requires a different number of arguments than the number passed to it. For example: Def Fna(X,Y)=X+Y Print Fna(J)
5	Illegal Statement	1. This can be caused by entering a line with a syntax error and then RUNning the program without correcting the line. 2. In certain systems, certain statements can be declared invalid. For example, POKE might be illegal in a multi-user system.
6	Print Item Size	An attempt was made to PRINT a single item which required more characters than the current page width. For example: Set 0,10 Print "Lots of characters"
7	Too Many Gosubs	Subroutines are nested within subroutines to a depth which exceeds that allowed by Basic.
8	Expression Too Complex	Too many levels of expressions, too many parentheses or function references.
9	Return, No Gosub Active	The program has no place to RETURN. This can be caused by deleting a line with a GOSUB statement and then encountering its corresponding RETURN statement.
10	Next Without For	FOR and NEXT statements must be paired. This error may occur if a line containing a FOR statement is deleted.

12	User Function not Defined	A user function is referenced by the program but has not been defined. If the line containing the function definition (DEF FNS(X)) is deleted, the function is no longer defined.
13	Invalid Dimensions given	Invalid argument(s) in the DIMension statement. For example: Dim A(-20) a negative number Dim B(5,5,5,5) too many subscripts Dim C$(20000) too large an integer (> 16382)
14	Goto or Gosub non-existent line	A GOTO or GOSUB statement refers to a line that does not exist.
15	Subscript Value(s)	The values assumed by subscripts must be less than those in the DIMension statement.
16	Number of Subscripts	The number of subscripts associated with a variable must match the number of subscripts in the DIMension statement.
17	Duplicate definition of label or function	An attempt has been made to give two different statements the same line label or to give two different functions the same function name.
19	Use of undefined line label	Control is transferred to a line label which does not exist.
20	Run time stack improperly nested	The terminating instruction of a control structure does not match the initial instruction of that control structure. For example: 10 While x 20 Enddo
21	Attempt to go back to altered or deleted line	A statement containing part of a control structure was EDITed or DELETEd and then an attempt was made to return control to that statement.

22	DIM would overflow top of existing COMMON	The maximum size of the Common Storage Area is defined in the main program (Partition zero).
23	Bad Begincommon/ Endcommon sequence	An ENDCOMMON instruction was encountered with no previous corresponding BEGINCOMMON instruction.
24	String/numeric expression mismatch	A syntax error for an expression incorrectly involving both string and numeric data. For example: If Ax$ = Bp$ + 7 Then 200
71	No such procedure available	PROCEDURE name not found in the current Partition or any other Partition or the Current Library (if open).
72	Bad arguments to a procedure CALL/ENDPROC	The arguments for a Procedure CALL or ENDPROC do not match the arguments in the PROCEDURE definition. For example: 10 Call .Xyz (Mat Aa) . . 500 Procedure .Xyz (String$) . .
73	No free partitions to load procedure/ module into	All eight Partitions are either manually or automatically LOCKed.
74	Invalid procedure library	The specified Library was not properly built. The Library must be composed of one or more SAVEd files which have been concatenated by the LIBBUILD program.
99	FEATURE NOT IMPLEMENTED	This feature has not been implemented.

101	End of Statement/ End of Line	This is an internal Basic error - please document and mail to Cromemco, Customer Service Dept.
102	Out of Memory	There is not enough memory to store the array (string) or to execute the specified control structure. Running or changing a program after the occurrence of Error 102 produces unpredictable results and should not be done.

USER TRAPPABLE (NON-FATAL) ERRORS

This section describes errors that can be trapped with the ON ERROR instruction described in Chapter 17.

-----Error-----

Number	Message	Meaning
128	File not Found	File not found on disk (file not in directory) or the device name is not in the device directory list.
129	Illegal Filename	An illegal file name was passed.
130	Invalid Command for Device	A command was given to a device which that device was incapable of performing. For example: a read command given to a line printer.
131	File Already Open	An OPEN command was given to a file which was already OPEN.
132	File Not Open	A read or write was attempted using a file which had not been OPENed.
133	File Number Out of Range	The file number requested was outside the allowable range. The file number must be greater than 0 and less than or equal to the maximum channel number. The file number can never be greater than 16.
134	Cannot Open File	A message from the device driver. (A non-zero value returned on OPEN.)

135	No File Space	All files in use. The system must have one unused channel to do a LIST, ENTER, SAVE, or LOAD. CDOS only - no more space on disk (or there are 64 directory entries).
136	File Mode Error	A read was attempted from a write only file or vice versa.
137	Cannot Create file	An attempt was made to CREATE a file that already exists.
138	File Read: No Data	End of file read, or, for random access only, an attempt to read a portion of the file which had not been written.
139	File Write	A message from CDOS - an attempt was made to write to a write protected disk or an error occurred while writing to the disk.
140	File Position/ status	An attempt was made to read a negative file record or record larger than 240K bytes.
141	No Channels Available	All I/O channels in use. (The maximum number of channels is system dependent.)
142	Cannot Close File	The specified file has been erased from the disk or a different disk has been inserted in the drive.
143	KSAM-Invalid Alternate File Request	An Alternate file request was given to a Primary file or the corresponding Primary file was closed or missing or a KALTCREATE or KALTOPEN instruction was given with an invalid Primary file reference.
144	KSAM-Key Length	The combined Key length for both the Primary and Alternate Keys is greater than 250 bytes or the Key length is shorter than originally specified (except KGETAPP).
145	KSAM- Record Size	The variable list in a read or write instruction was greater than the specified record size.

146	Not a KSAM file	A Basic-KSAM operation was attempted on a standard (non-KSAM) file.
147	File is KSAM file only	A standard operation was attempted on a Basic-KSAM file.
148	Wrong Number of Volume Names Specified	The number of files specified on a KOPEN or KALTOPEN must match the number of files specified by the corresponding KCREATE or KALTCREATE instruction plus any files which have been added by use of the KADDVOL instruction.

NOTE: THERE IS A DETAILED DESCRIPTION OF ERRORS NUMBERED 161-190 AT THE END OF CHAPTER 22.

161	KSAM-Primary File Empty	An attempt to access a record of an Empty Primary Data File.
162	KSAM-Primary Beginning of File	An attempt to read a record before the first record of the Primary Data file.
163	KSAM-Primary End of File	An attempt to access a record beyond the last record of the Primary Data File.
164	KSAM-Primary Invalid Key	The Primary Key already exists (write operations) or does not exist (read, update, and delete operations.)
167	KSAM-Primary Invalid Request	Refer to the detailed descriptions of error codes in the Basic-KSAM chapter of this manual.
168	KSAM-Primary Invalid Create Parameters	The parameters in the KCREATE instruction are incorrect.
169	KSAM-Primary File exists	A file with the same file name and extension already exist on the specified drive.
170	KSAM-Primary Key Set Full	The Primary Key Set is Full. The file must be closed and compacted using the utility program before proceeding.
171	KSAM-Primary No Free Blocks	No more space on disk. Either add another volume (KADDVOL) or close and compact the file (utility program) before proceeding.

173	KSAM-Primary Open Failed	The operating system could not open the Primary Data file.
174	KSAM-Primary I/O Error	The operating system could not access the disk as was required.
177	KSAM-Alternate File Empty	An attempt to access a record of an Empty Alternate Key File.
178	KSAM-Alternate Beginning of File	An attempt to read a record before the first record of the Alternate Key file.
179	KSAM-Alternate End of File	An attempt to access a record beyond the last record of the Alternate Key File.
180	KSAM-Alternate Invalid Key	The corresponding Primary Record does not exist (read, update, and delete operations.)
183	KSAM-Alternate Invalid Request	Refer to the detailed descriptions of error codes in the Basic-KSAM chapter of this manual.
184	KSAM-Alternate Invalid Create Parameters	The parameters in the KALTCREATE instruction are incorrect.
185	KSAM-Alternate File exists	A file with the same file name and extension already exist on the specified drive.
186	KSAM-Alternate Key Set Full	The Alternate Key Set is Full. The file must be closed and compacted using the utility program before proceeding.
187	KSAM-Alternate No Free Blocks	No more space on disk. Either add another volume (KADDVOL) or close and compact the file (utility program) before proceeding.
189	KSAM-Alternate Open Failed	The operating system could not open the Alternate Key file.
190	KSAM-Alternate I/O Error	The operating system could not access the disk as was required.

200	Invalid Hex Number	Hexadecimal numbers must contain only the characters 0 through 9 and A through F.
201	Integer Overflow	A value greater than 32767 was assigned to an integer variable.
202	Function Argument Value	A function was called using an illegal argument. For example: Sqr(-2).
203	Invalid Input	An attempt was made to INPUT non-numeric data into a numeric variable.
204	Too Much Input	An attempt was made to INPUT more items than were called for in the INPUT instruction.
205	Not Dimensioned	A reference was made to a subscripted variable which had not been DIMensioned.
206	No Data Statement	An attempt was made to READ past the end of the DATA supplied. Either there was a READ with no DATA statement or there were not as many items in the DATA statement as in the READ list.
207	Data Type Mismatch	An attempt was made to READ a numeric value to a string variable or vice versa. For example: 10 Data 5 20 Read A$
208	Number Size	An attempt was made to assign a value outside of the range 9.99E+62 to 9.99E-65 to a variable.
209	Line Too Long	A line longer that 132 characters was entered.
210	Input Timeout	See the SET instruction for information about this error.
250	Overflow/ Underflow	A floating point operation produced a number outside of the range 9.99E+62 to 9.99E-65. For example: A=1/0. Or, Integer arithmetic caused results outside of the range -32768 to 32767.
251	Errproc return from a procedure	This error is set when an ERRPROC instruction is executed.

Appendix A

ASCII CHARACTER CODES

This table lists the ASCII codes.

DEC.	HEX	CHAR.	DEC.	HEX	CHAR.	DEC.	HEX	CHAR.
000	00	NUL (CONTROL-@)	043	2B	+	086	56	V
001	01	SOH (CONTROL-A)	044	2C	,	087	57	W
002	02	STX (CONTROL-B)	045	2D	-	088	58	X
003	03	ETX (CONTROL-C)	046	2E	.	089	59	Y
004	04	EOT (CONTROL-D)	047	2F	/	090	5A	Z
005	05	ENQ (CONTROL-E)	048	30	0	091	5B	[
006	06	ACK (CONTROL-F)	049	31	1	092	5C	\
007	07	BEL (CONTROL-G)	050	32	2	093	5D	]
008	08	BS	051	33	3	094	5E	^
009	09	HT	052	34	4	095	5F	<
010	0A	LF	053	35	5	096	60	'
011	0B	VT	054	36	6	097	61	a
012	0C	FF	055	37	7	098	62	b
013	0D	CR	056	38	8	099	63	c
014	0E	SO (CONTROL-N)	057	39	9	100	64	d
015	0F	SI (CONTROL-O)	058	3A	:	101	65	e
016	10	DLE (CONTROL-P)	059	3B	;	102	66	f
017	11	DC1 (CONTROL-Q)	060	3C	<	103	67	g
018	12	DC2 (CONTROL-R)	061	3D	=	104	68	h
019	13	DC3 (CONTROL-S)	062	3E	>	105	69	i
020	14	DC4 (CONTROL-T)	063	3F	?	106	6A	j
021	15	NAK (CONTROL-U)	064	40	@	107	6B	k
022	16	SYN (CONTROL-V)	065	41	A	108	6C	l
023	17	ETB (CONTROL-W)	066	42	B	109	6D	m
024	18	CAN (CONTROL-X)	067	43	C	110	6E	n
025	19	EM (CONTROL-Y)	068	44	D	111	6F	o
026	1A	SUB (CONTROL-Z)	069	45	E	112	70	p
027	1B	ESC (CONTROL-[)	070	46	F	113	71	q
028	1C	FS (CONTROL-\)	071	47	G	114	72	r
029	1D	GS (CONTROL-])	072	48	H	115	73	s
030	1E	RS (CONTROL-^)	073	49	I	116	74	t
031	1F	US (CONTROL-)	074	4A	J	117	75	u
032	20	(SPACE)	075	4B	K	118	76	v
033	21	!	076	4C	L	119	77	w
034	22	"	077	4D	M	120	78	x
035	23	#	078	4E	N	121	79	y
036	24	$	079	4F	O	122	7A	z
037	25	%	080	50	P	123	7B	{
038	26	&	081	51	Q	124	7C	|
039	27	'	082	52	R	125	7D	}
040	28	(	083	53	S	126	7E	~
041	29	)	084	54	T	127	7F	DEL
042	2A	*	085	55	U			

Legend

LF=Line Feed
FF=Form Feed
CR=Carriage Return
DEL=Rubout
ESC=escape character

Reader Responses To This Documentation

Dear Reader,

We have made a sincere effort to provide you with the information you need in this manual. If you should find the documentation deficient or in error, let us know so we can correct it. We appreciate and value your response; it will be useful in improving the documentation. Please detach and use the Reader Response Card below to send us your comments.

Thank you for your time and interest in Cromemco products.

Sincerely,

Winthrop A. Stiles III

Technical Publications Manager

(Detach Here)

Cromemco®

Reader Response Card

To: Winthrop A. Stiles III,
Technical Publications Manager

Re (Manual title):_____

My System is (Specify configuration):_____

The following information is incorrect (Please specify page number):_____

_____

_____

_____

_____

_____

_____

(Detach Here)

(Fold Here)

The following additional information would be helpful:_____

_____

_____

What general suggestions do you have for improving this manual?_____

_____

_____

If you need a response from Cromemco, please print your name, mailing address, and telephone number:

Name:_____

Address:_____

_____

_____

Telephone: (____)_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 599 MOUNTAIN VIEW, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Cromemco®

Attn: Winthrop A. Stiles III
Technical Publications Manager
280 Bernardo Avenue
P.O. Box 7400
Mountain View, CA 94039

(Fold as indicated and tape the edge)

Cromemco®

280 Bernardo Ave.
P.O. Box 7400
Mountain View, CA 94039