CROMEMCO 16K EXTENDED BASIC USER'S MANUAL

Cromemco, Inc.

280 Bernardo Avenue

Mountain View, CA 94040

Part No. 023-0037 Revision 2.0

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

Chapter 2 (cont'd) Page No.

TABLE OF CONTENTS (continued)

Chapter 2 (cont'd) Page No.

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

1.0 INTRODUCTION: DESIGN CRITERIA OF CROMEMCO 16K EXTENDED BASIC

Many different versions of BASIC are commercially available from computer manufacturers. These different versions may vary greatly in computational precision, programming power, ease of programming, speed of execution, and amount of memory required for the BASIC program. For example, there are a number of BASICs available which are similar in capability to the original BASIC developed at Dartmouth College. These BASICs typically require about 8K of memory. There are also a number of "tiny" BASICs available that require only 2K of memory but which have different or less capability than an 8K BASIC. For many of today's demanding applications, more features are required than were provided in the original Dartmouth BASIC. A BASIC that provides advanced features and capabilities is frequently called an Extended BASIC. The features included in Extended BASICs may also vary widely among computer manufacturers.

Cromemco 16K Extended BASIC (which was developed exclusively for Cromemco by Shepardson Microsystems, Inc.) was designed to maximize computational precision, programming power, and speed of execution by fully utilizing the extensive 158 instruction set of the Z-80 microprocessor.

Cromemco's powerful Extended BASIC was specifically designed to meet the most demanding requirements of business

firms (such as inventory and accounting programs) while providing the flexibility and speed necessary for real-time control applications in both industry and the home. A brief review of some of the features included in Cromemco 16K Extended BASIC follows.

One major feature of Cromemco 16K Extended BASIC is rapid, 14-digit arithmetic using the powerful, binary-coded decimal (BCD) arithmetic instructions which are unique to the Z-80 microprocessor. All arithmetic operations are performed using the 16 high-speed registers of the central processor for intermediate storage. The speed and precision obtained when performing BCD arithmetic cannot be obtained on 7-register microprocessors such as the 8080. Many BASICs were designed for six- or seven-digit precision using computer-oriented binary arithmetic. It is possible to do calculations which are as accurate using binary arithmetic as they are using BCD arithmetic. However, many BASICs which use binary arithmetic have introduced highly visible errors when calculating dollar and cent conversions in business and financial programs. These conversion errors cannot occur in BASICs which utilize BCD arithmetic. In addition to arithmetic operations, all functions in Cromemco 16K Extended BASIC are computed to 14-digit accuracy. Many other BASICs that provide 14-digit accuracy do so at greatly reduced execution speeds and often without carrying full accuracy through all subroutines.

Real-time control applications often require integer, 16-bit (4½ digit) arithmetic. Cromemco 16K Extended BASIC provides this capability along with direct memory, input/output, and machine-code subroutine access through BASIC commands.

There is also often a need in real-time control system applications to store large arrays of data. Cromemco 16K Extended BASIC offers the flexibility of allowing the user to manipulate, store, and retrieve six-digit floating point numbers. This abbreviated form of the highly efficient, 14-digit mathematics capability is also slightly faster in execution than the 14-digit format.

Furthermore, in Cromemco 16K BASIC (unlike most micro-computer BASICs), constants used in program lines cause execution speeds as fast or faster than variable references. In addition, integer constants occupy no more room than a variable reference.

Execution speed is maximized in 16K Extended BASIC by the utilization of a semi-compiling design. The semi-compiling design provides many of the important features of both a compiler and an interpreter. In other words, the properties of a compiler which provide high execution speed are combined with the interactive capabilities and programming ease provided through an interpreter. For example, programs may be stopped during execution, examined, and modified. The computer may then be instructed to continue to run the program without affecting the proper execution of the program. The ability to stop, examine, and modify programs can greatly reduce the time and cost associated with custom programming.

This interactive capability is not available with optimized compilers. The semi-compiling operation of 16K Extended BASIC is performed after each program line is entered. Any obvious errors are detected at the time of entry rather than during execution. Consequently, the time required to enter and debug a program is reduced substantially.

Some versions of BASIC have either rudimentary or no error message capability. With these versions, if a "fatal" error causes the program to "crash" or halt unexpectedly during execution, it is up to the user to reconstruct the failure within the program. Other versions will produce a numerical value after the program fails. This value must be interpreted on a list of error messages. Some more powerful BASIC programs will return an English-language statement explaining the failure "after the fact". Cromemco 16K Extended BASIC not only generates English-language error messages, but it also examines each statement as it is entered and prints error messages immediately. The value of such pre-checking, using the error messages, can be pronounced. This capability is especially important when doing corrections to real-time control systems where inadvertent errors in modifications can be costly. In general programming, the interactive aspect of the error messages is extremely convenient and can substantially reduce programming time and errors.

Dynamic error trapping is provided in Cromemco 16K Extended BASIC for specialized routing of program flow in response to a BASIC-generated error message. For example, using the error trapping feature, overflow and underflow arithmetic errors can be handled without operator assistance. The value of this feature in preparing programs that will be used by unskilled operators cannot be underestimated. For instance, should a disk read/write error occur, 16K Extended BASIC can be programmed to type out a message such as "CALL MR. HIGGINS (415-030-1234) IN THE ACCOUNTING DEPARTMENT", thereby alerting skilled personnel who can correct the problem.

A TRACE feature is provided in Cromemco 16K BASIC to help determine the origin of errors which are difficult to find in a program being written. Line numbers are logged as they are executed with this feature. FOR loops are automatically indented as they are logged in the TRACE mode.

Cromemco 16K Extended BASIC also has advanced floppy disk input/output capabilities. For example, both programs and data may be stored. Programs may be stored in binary or ASCII format. Numeric and string data may be stored on disk using sequential or random access methods. The number of files which may be opened simultaneously is limited only by the amount of memory the user allocates for file purposes, and an unlimited number of files may be opened and closed during the course of a program. Chaining of program segments too large to fit in

memory at one time can be performed with variables passed through disk files. File names can be string variables and can be changed dynamically by the BASIC program. BASIC programs can be written, modified, or run by other BASIC programs. Line numbers can be altered and two programs can be called off disk and concatenated into one program. The versatility provided through the interaction of Cromemco's CDOS disk operating system and the disk input/output capabilities of Cromemco 16K Extended BASIC is uncommon.

Many BASICs have string handling capability. However, many of these limit the number of characters that can be stored in the string to a maximum of 64. In Cromemco 16K Extended BASIC, up to 32,768 characters or the available memory space are allowed -- whichever limit is smaller. Substrings are utilized to rapidly manipulate these strings. Again, the design of the language emphasizes speed, versatility, and accuracy. A special, high-speed, string-searching function is provided to increase throughput in string manipulation and sorting operations (such as are commonly found in mailing list processing programs).

Program listings from 16K BASIC are print justified and are uniformly spaced automatically for reading ease and clarity of documentation. FOR loops are also automatically nested and indented by the listing function. In the semi-compiling operation, the only statements that are directly stored in

memory at entry time are the REM comment statements (and even here spaces are compressed). All other statements are semi-compiled and stored in machine-readable format. When listed, this machine-readable format is reconverted into ASCII format and then printed. Thus all spacing and other variations are lost in the conversion process. This conversion process allows Cromemco 16K BASIC to efficiently store large programs by storing programs only once in semi-compiled format. In contrast, many other BASICs always maintain an uncompiled "one-to-one" representation of what was typed in, along with any internal, execution-related storage. Such BASICs require extra program storage if blank spaces are typed inadvertently into the program text. Blank spaces can slow down program execution. Cromemco 16K Extended BASIC does not store, nor is it affected by, inadvertently typed blank spaces. Furthermore, these extra blank spaces are not listed in the program output.

There are a number of features commonly found in most Extended BASICs that are included in Cromemco 16K Extended BASIC. These features include multi-statement lines; one, two, and three dimensional arrays; 26 user-definable functions; advanced formatting capabilities with PRINT USING, TAB, SPACE, and SYS functions; and direct machine language interaction with INP, OUT, PEEK, POKE, and USR commands. Finally, Cromemco 16K Extended BASIC also has flexible INPUT and PRINT routines which are capable of supporting a wide

range of terminals and line printers.

This manual was designed for use both by users who have no prior experience with BASIC and by experienced programmers. Users with no prior experience are encouraged to read through all the sections in Chapter 1. These sections include introductory background material and material designed to show the inexperienced user how to input a BASIC program on a terminal. The experienced programmer can skip these sections and proceed directly to Chapters 2 and 3. Chapters 2 and 3 contain descriptions of all commands available in Cromemco 16K Extended BASIC.

1.1 BACKGROUND: COMPUTERS AND COMPUTING POWER

A computer is a device which performs high-speed mathematical or logical calculations or which processes information derived from coded data in accordance with a predetermined program. This predetermined program is a set of instructions arranged in logical sequence which directs the computer to perform a desired operation.

All computers are based on one fundamental concept -- the binary digit. The binary number system uses only the digits 0 and 1. What makes the binary digit concept so useful is that it can be represented by any device which is capable of assuming one of two possible stable states. A computer, even in its most complex form, is simply a collection of logic gates which respond to the presence or absence of a voltage. The presence or absence of such voltages, called "logic levels", makes it possible to represent binary conditions in a logic circuit. Typically, these logic levels are defined as 0's for off states and 1's for on states.

Binary digits, commonly called bits, can be used to represent numbers of any magnitude by grouping digits together. The primary level at which binary digits are grouped within a particular computer is called word size. Most microcomputers use an 8-bit word size. This 8-bit unit is called a byte. If we consider the various computer

words we can make out of all possible combinations of eight bits, we find that there are $2^8=256$ possible combinations. We can then build a computer which will respond differently to each byte which we present as an instruction. For example, an instruction of all 0's might mean stop, while 10101010 might mean increment a number by 1. Furthermore, a particular byte might stand for one operation when used with one model of computer and might stand for a very different operation in another computer system. The important point is that all computers execute instructions through this binary "language" which is a combination of on-off states represented by 0's and 1's.

HIGH LEVEL AND LOW LEVEL LANGUAGES

As stated in the introduction, a computer must be programmed to perform calculations or to process information. Programs are prepared in a "language" with precise rules of grammar. Currently available computer languages fall into one of two major categories: high-level or low-level languages. Internally, computers perform various operations according to the binary language described earlier. This binary language is commonly called "machine language". Machine language is at the lowest level of computer languages and consists of groups of on and off states. A machine language word is any combination of eight 0's and 1's, e.g. 01100100. As you can see, it is nearly impossible to tell what a given machine language word means by simply looking at the word. Furthermore, programming a computer in this fashion is both time-consuming and tedious.

To simplify the programming process, some easier method of entering and recognizing instructions is required. The next step up in low-level languages then is an "assembly language" which uses short mnemonic symbols that assist in the definition of the instruction. An assembly language translates simple commands, such as HALT or INCREMENT, into the appropriate machine language instructions. But even this form of programming is too cumbersome for many applications. In assembly language, a program to print out "HELLO" on a terminal may require as much as 50 steps. Furthermore,

individuals without a great deal of computer expertise often find it difficult to translate problems into a computer program at either the assembly or machine language level.

As a result, higher level languages were developed with commands and rules of grammar which are easier to learn and apply. Problem-oriented, high level languages include FORTRAN, COBOL, ALGOL, PL1, and a variety of other languages. These high-level languages are translated into machine language through a program called a compiler. Any language that must be compiled or translated one or more times before it is reduced to machine code is said to be a high-level language.

BASIC AND ITS USES

Most of the high-level languages mentioned in the previous section were designed for use by special groups comprised of relatively sophisticated computer users. Consequently, most of these languages are somewhat difficult to learn. However, as the computer moved from the design laboratory into the university laboratory and then on into business, home, and school usage, the need for an easy-to learn, grammatically simple, English-language-based, programming language grew. To answer this need, Professors John G. Kemeny and Thomas E. Kertz of Darthmouth College developed the first version of BASIC in 1965. BASIC is a higher level language which is easy to learn and which can be applied to most computing problems.

In addition to being easy to learn, BASIC has a number of features which make it useful for two very important and common applications: time-sharing and interactive programming. In BASIC, simple, English-language commands are translated line-by-line as the program executes into machine code through a program called an interpreter. This interpreter makes it easy to interrupt the conversion process, proceed to another task, and then return to pick up where you left off. Consequently, a computer can be used to allow multiple users running BASIC to "timeshare" a system. In time sharing, the computer handles each user in some specified sequence. The high speed of the computer, however, makes it

appear that all users are being handled simultaneously.

BASIC is also particularly well-adapted for use in interactive programs which require continual user interaction with the computer. Interactive programming allows the user to continually update and revise a program based on intermediate results. For complex problems involving a number of parameters that may require changing before a solution is reached, the interactive programming capability is a powerful tool.

Because of these important features, the library of programs written in BASIC -- many of them in the public domain -- has expanded rapidly. Many complex business programs, including inventory control, payroll, shipping and receiving, and accounting, are available in BASIC. There are programs written for computer-assisted instruction (CAI) and for a wide range of interactive games in which users compete with each other or with the computer in simulations of a variety of situations and events.

In addition, the same computer system may be used to streamline accounting procedures, control manufacturing processes, regulate energy usage, or play games. Only the BASIC program need be changed. With the introduction of inexpensive, microcomputer-based, time-sharing systems (such as the Cromemco Z-2 Educational Time-Sharing System), true multi-user computer operation becomes available at only a fraction of its cost five years ago.

1.2 A GLOSSARY OF TERMS USED IN 16K BASIC

ARGUMENT

An argument is an independent variable used with a BASIC word whose value can be specified by the user to instruct BASIC to perform a certain task. For example, in the statement:

```
PRINT A, B
```

A and B are arguments to the BASIC word 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, and various symbols. The ASC (X) function will return the ASCII equivalent of any argument. A table of ASCII code is provided for reference in the Appendix.

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 ERR
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.

A powerful feature of Cromemco 16K 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.

CURRENT PROGRAM

The current program is any program with which the user is currently interacting. When 16K 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. When the user edits a program, it remains the current program.

DATA

The term data 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 DATA statement.

DEFAULT

With certain BASIC words, an argument may be added optionally 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 value for the statement line

```
RENUMBER
```

is 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 statement line

```
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 disk is similar in appearance to a phonograph record. Most microcomputer systems currently offer disk storage capabilities through either large floppy disks or through mini floppy disks. The floppy and mini floppy terms refer to the two different sizes (8" and 5" respectively) of the flexible plastic disks used with the disk assemblies.

EXPRESSION

An expression is defined as any combination of variables, constants or operations which is evaluated as a single value or logical condition. For instance, in the statement:

```
10  LET A = (B*C) + (A*D)
```

the (B*C) + (A*D) operation, which is equated with variable A, is interpreted as an expression. In the statement:

```
10  IF A = B THEN GOTO 250
```

the logical comparison A = B is called an expression.

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 ROM. For example, the BASIC interpreter might be considered to be firmware.

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 16K 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 metal (or "hard") elements of a computer system. Items such as printers, terminals, and the computer itself are considered to be hardware.

INTEGER

An integer is simply 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, disk drives, and cassette tapes are examples of devices that accept input data from the user, another peripheral device, or from the computer memory, and that output data from the computer. Data displayed on a terminal may be output from a disk or a cassette or the data may be output from computer memory or from a peripheral memory device.

MATRIX

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

$$\begin{matrix} 3 & 2 & 0 \\ 1 & 4 & 6 \\ -3 & 4 & 5 \end{matrix}$$

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

$$\begin{matrix} a_{11} & a_{12} & a_{13} & \cdots a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots a_{2n} \\ \vdots & & & \\ a_{m1} & & & \cdots a_{mn} \end{matrix}$$

The individual entries in the matrix are called elements. For example, the quantity a_{ij} 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., M_{ij}) or three (i.e., M_{ijk}) 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.

The respective formats for declaring the number of rows and columns in one, two, and three dimensional arrays are:

```
10  DIM A(n)
20  DIM B(n,m)
30  DIM C(n,m,r)
```

where n, m, and r are integers.

Note that 16K BASIC programs execute significantly faster if variables used for subscripting (and/or loops) are declared as type INTEGER. Also, using the Øth element of matrices (and strings) can save memory space.

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, memories consist of an array of bistable, individually addressable elements which each represent a single binary digit. Information can be stored either in "main memory", which commonly consists of RAM or ROM, or in external storage devices, which include disks, magnetic tape, and magnetic drums.

PERIPHERAL DEVICE

Peripheral devices are units which are used in conjunction with a computer but which are external to the computer. 'Peripherals' refers 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.

PROGRAM

A computer program is a set of commands arranged into statement lines. The commands 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 is designed to program such 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. This convention is part of the 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 these 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.

SOFTWARE

Software is a term used to refer to the programs, languages and procedures used in a computer. For example, the 16K BASIC language and any BASIC program are identified as software.

STATEMENT

A statement in BASIC is an instruction to the computer. A statement is generally defined as any line in a BASIC program which is preceded by a number. For example:

```
100  A = B*C
```

is defined as a statement. Typically, a statement can contain a maximum of 132 characters.

A powerful feature of Cromemco 16K 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.

STATEMENT LINE NUMBER

All lines in BASIC begin with a line or statement number. For example:

```
10  PRINT A, B
```

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 is used to identify lines which require modification or deletion from the program.

STRING

A string is a sequence of alphanumeric characters, spaces, and special characters. In 16K BASIC, strings are enclosed within quotation marks. Examples of valid strings include:

```
"CROMEMCO 16K BASIC"
"12345"
"THIS PROGRAM PRINTS SQUARE ROOTS"
```

The statement:

```
100  PRINT "CROMEMCO 16K BASIC"
```

will output the string

```
CROMEMCO 16K BASIC
```

STRING VARIABLE

A string variable may be used when the user wishes to assign a string value to a variable. String variables consist of a single letter of the alphabet (A through Z), or a single

letter and one digit (0 through 9), followed by a dollar sign ($). Examples of valid string variables include:

B$

C3$

D8$

VARIABLE

A variable is a quantity that can assume any one of a given set of values. In 16K BASIC, variables are defined by a single letter (A through Z) or a single letter followed by one digit (0 through 9). Examples of legal variable names include:

A

A1

C

C0

Variables represent numeric values. In the statement:

```
20  A = 8 + 2
```

A is the variable and 8+2 or 10 is the value of A. A new value can be assigned to A at any subsequent point in a program.

1.3 PROGRAMMING FORMATS FOR 16K BASIC

Cromemco 16K BASIC includes about 60 BASIC instructions. Some examples of common BASIC instructions are listed below:

LIST	FRAction
GO to SUBroutine	RaNDom number

In each case, the capitalized letters form the BASIC instruction.

Program Directive Statements

One class of BASIC words consists of Program Directive Statements. These statements instruct the computer to perform operations within the context of the program itself. The instructions contained in a statement line are not executed until the program is run.

Several different formats are used for Program Directive Statements. For instance, if the computer system encounters the statement line:

```
30  INPUT A
```

in the course of program execution, it will type a question mark (called in this context a 'prompt') onto the terminal and await a response from the user. The proper response is a numerical value. Any response other than a number will cause the computer to issue an error message.

The computer assigns the input value to A.

When the computer executes the statement line:

```
40 PRINT "CROMEMCO 16K BASIC"
```

it prints the text enclosed within the quotation marks onto the terminal or other file device. Variables and string statements need not be enclosed within quotes. To print the value of variable A onto the terminal, a simple:

```
50 PRINT A
```

instruction is all that is needed.

Other Program Directive Statements provide elaborate routing of program control and extensive manipulation features. A complete description of each program statement is given in Section 2.1 and 2.2.

Program Command Instructions

A second class of BASIC words, called Program Command Instructions, controls the execution of entire programs. With these commands, a program may be RUN, STOPped, LISTed to a terminal or other file device, ENTERed into user memory, CONtinued, RENUMBERed, or otherwise manipulated.

These instructions are used when a current program is active, as opposed to the text input mode in which no specific program has been created or called from memory. The text

input mode applies when the computer system is first turned on, or when a previous program has been STOPped or SCRatched. The double "greater than" marks (>>) are used to indicate that the computer is ready to accept new input. If the computer is ready to accept program input or directives, the ">>" symbols will be printed flush with the left margin of the printer or CRT screen.

After the >> symbol has been generated by the computer, a line or program command instruction may be typed in on the terminal. For example, the following command instructs the computer to list out the current program:

```
>> LIST
```

As a rule, this will be the format for most Command Statements. The LIST, RENUMBER, and AUTOL commands may also be used with delimiting line numbers. Consult the definition of a given BASIC command to determine the correct format.

Programmed Functions

Cromemco 16K BASIC includes a number of pre-programmed functions which perform common, frequently used calculations. The available functions include arithmetic, trigonometric, boolean, string handling, matrix handling, assembly language subroutine, and system functions. In addition, Cromemco 16K BASIC permits the user to define up to 26

functions. For example, one of the arithmetic functions available in BASIC is the square root of a numeric expression. The format for this function is:

```
10 A = SQR (X)
```

where X is any numeric expression. This statement line instructs the computer to assign the value equal to the square root of X to the variable A.

String Control Statements

Cromemco 16K BASIC manipulates alphabetic information through strings and string variables. A string is defined as any combination of characters, including letters, numbers, special characters and spaces, but excluding quotation marks. A string literal is defined as any string enclosed in quotation marks. For example, in the following statement:

```
10  PRINT "THIS IS A STRING"
```

the phrase "This is a string" is a string literal.

String variables are also used in Cromemco BASIC. A string variable is defined as any letter A through Z followed by a dollar sign or any letter and any number Ø through 9 followed by a dollar sign. The following are valid string variables:

```
A$
F2$
Z$
```

Any string can be assigned to a string variable. There are a number of instructions available in BASIC which facilitate string manipulation. Descriptions of these instructions are included in the following sections.

Printing Formats

Cromemco 16K BASIC also provides advanced print formatting capabilities with format control commands such as PRINT, TAB, SPC, and PRINT USING. The PRINT USING statement, in particular, allows the user to define special formats for the output of numeric data. This format may specify the number of digits following a decimal point, the printing of a dollar sign before a number, spacing between numbers, and a variety of other formatting options.

Numerical Representation in 16K BASIC

In Cromemco BASIC, any or all variables may be set to integer, short floating point, or long floating point mode. The default mode is the long floating point mode.

In the long floating point (LFP) mode, numbers of up to 14 digits are represented in a decimal format (e.g., 3.1415926543287). In the short floating point (SFP) mode, numbers are limited to 6 digits (e.g., 3.14159).

If there are more digits than the mode allows in the expression of a numerical value, the number is represented in

scientific notation. For example, in short floating precision:

Ø.ØØØØ1 becomes 1E-Ø5

12340000 becomes 1.234EØ7

To set all variables within a program to a given mode, one of the following commands is given at the beginning of the program:

IMODE (Integer Only)

LFMODE (Long Floating Point)

SFMODE (Short Floating Point)

In addition, any given variable may be formatted to one of these modes through similar statements. For example:

```
10  INTEGER A
20  LONG B
30  SHORT C
```

In this example, the variable A is in integer-only mode, the variable B is in long floating point mode, and the variable C is in short floating point mode.

Long floating point numbers occupy 8 bytes, short floating point occupy 4 bytes, and integer numbers occupy 2 bytes.

These specific mode set instructions (i.e., INTEGER A) override the general mode set instructions, so care should be taken in the use of these statements.

For example, in the following program the SFMODE is set but the variable A is set to INTEGER mode within the program:

```
SFMODE
10  INTEGER A
20  A=2.333
30  PRINT A
40  END
RUN
2
***40 END***
```

In this example, statement 10 overrides the general SFMODE command. If statement 10 is deleted, the program will run as follows:

```
SFMODE
20  A=2.333
30  PRINT A
40  END
RUN
2.333
***40 END***
```

Special note should be made of the two types of constants in 16K BASIC; floating and integer. When a constant which is less than 10,000 is entered without a decimal point, it is understoood to be an integer constant. For example:

```
10  A = 2/3
20  PRINT A
30  END
RUN
Ø
***30 END***
```

The Ø is the result of integer arithmetic being performed on the expression.

A decimal point can be used to override the integer mode. In that case, the program above can be written as follows:

```
10  A = 2./3.
20  PRINT A
30  END
RUN
.66666666666667
***30 END***
```

This is the result of floating point arithmetic. All integers greater than 10,000 go to the defined mode (i.e., short floating point or long floating point). A simple rule to follow is "when in doubt use the decimal point in constant representations."

Also, note that programs will execute significantly faster if variables used in loops and/or for subscripting are declared as type INTEGER.

HEXADECIMAL NUMBERS

In Cromemco 16K BASIC, a hexadecimal number may be used anywhere a constant is called for (i.e., expressions, INPUT statements, etc.). The format for hexadecimal numbers is:

%h%

where h is any hexadecimal number. For example: if h=ØØFØ, then this number can be represented in 16K BASIC as %ØØFØ% . The user must supply the leading and trailing % sign.

ARITHMETIC OPERATIONS

A number of special characters are used in 16K BASIC to indicate arithmetic and relational operations. These characters, along with their definitions, are listed below:

Character	Defintion
+	Plus Sign (used to indicate a positive number)
-	Minus Sign (used to indicate a negative number)
↑ or **	Exponentiation
/	Division
*	Multiplication
+	Addition
-	Subtraction
=	Equal
<	Less Than
< =	Less Than or Equal To
>	Greater Than
> =	Greater Than or Equal To
< >	Not Equal

The last six characters represent relational operators which are used to compare two expressions. The first seven characters represent arithmetic operations.

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, exponentiation, division and multiplication (these 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 the normal left to right priority.

Example:

```
(X + Y - (Z1 * Z2))/2
```

In the example above, the value of Z1*Z2 is calculated first. Then the value of X is added to the value of Y and the value of Z1*Z2 is subtracted from this value. The total is then divided by 2.

BOOLEAN OPERATORS

Four Boolean operators, AND, OR, NOT, and XOR, are used in Cromemco 16K BASIC. The result of each of these operations is dependant on the value(s) of one or two logical variables. A logical variable is one which can take on one of two values: 1 (=true) or 0 (=false).

AND

The AND 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 Ø.

OR

The OR operator compares two logical values and if either or both are equal to 1 then the result is 1. Otherwise, the result is Ø.

XOR

The XOR operator returns a Ø if the logical values are identical and a 1 if the logical values are not identical.

NOT

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

Note: All variables which are not equal to zero are considered to be "true" (=1) when used with Boolean operators.

1.4 LOGGING ON WITH 16K BASIC

This section outlines the steps required to input and run a BASIC program on a computer system. The most common input devices are cathode ray tube (CRT) terminals and teletypewriters (TTY). When using the Cromemco Z-2 microcomputer system, either device can be plugged directly into the I/O connector sockets in the rear of the machine. (Note: On TTY terminals, there is a switch which should be set to LINE.) The Cromemco ZPU provides power-on jump circuitry to begin automatic program execution when power is turned on. The method used to enter 16K BASIC is dependent on the configuration of your machine. The next five pages describe how to enter 16K BASIC from DISK, PROM, and Paper Tape.

Note: When BASIC is loaded into a minimum DISK system configuration with 32K bytes of memory, the space available to the user should be 3.8K bytes. The other 28.2K bytes are occupied by BASIC, CDOS I/O routines, etc. See notes at the end of the manual for ways to increase the user space.

(Appendix C: Changing the Number of I/O Channels)

NOTE: It is important to remove disks from the disk drive before turning the power off or on. This will eliminate the possibility of stray bits being written on the disk as the system is powered up or down.

CROMEMCO 16K BASIC FROM DISK

Memory Requirements:

1. at least 32K RAM starting at location Ø

System Set Up:

1. set the POWER ON JUMP switch on the ZPU board to CØØØH

   Switch A15=1 — 2^{15}

   " A14=1 — 2^{14}

   " A13=Ø — 2^{13}

   " A12=Ø — 2^{12}.

2. set the four switches on the 4FDC board as follows

   Switch 1='OFF'

   " 2='OFF'

   " 3='ON'

   " 4='OFF'

Method:

1. insert the CROMEMCO 16K BASIC disk in drive A (on the left).
2. depress the return key on the console several times so that CDOS can determine the baud rate of your terminal.
3. CDOS will return with a prompt {A.}
4. type {BASIC} and a {carriage return}
5. CROMEMCO 16K BASIC will respond with a prompt {>>} to indicate that it is ready to accept a command.

CROMEMCO 16K BASIC FROM PROM

Memory Requirements:

1. 16K PROM starting at location 8ØØØH

2. at least 8K RAM starting at location Ø

System Set Up:

CROMEMCO 8K BYTESAVER & 8K BYTESAVER II PROM boards

1. turn program power switch 'OFF'

2. turn program enable switches 'OFF'

3. plug the 16 CROMEMCO 16K BASIC PROMs into the two boards in numerical order

board one - 8ØØØ into PROM socket Ø
8ØØØ into PROM socket 1
etc.

board two - AØØØ into PROM socket Ø
AØØØ into PROM socket 1
etc.

4. set LOGICAL ADDRESS BLOCK SELECT to 8ØØØH & AØØØH

BYTESAVER jumper wires:

board one - A15 HI, A14 LO, A13 LO
board two - A15 HI, A14 LO, A13 HI

BYTESAVER II switches:

board one - 15=1, 14=Ø, 13=Ø
board two - 15=1, 14=Ø, 13=1

16KPR board

1. set LOGICAL ADDRESS BLOCK SELECT to 8ØØØH

(switch 15=1, switch 14=Ø)

2. plug the 16 CROMEMCO 16K BASIC PROMs into the board in numerical order

8ØØØ into PROM socket Ø
8ØØØ into PROM socket 1
etc.

All systems

1. set jump start address on the ZPU board to 8ØØØH

   Switch A15=1
   " A14=Ø
   " A13=Ø
   " A12=Ø

2. select bank Ø on the PROM board (switch 8 on)

Method:

1. depress the return key on the console several times so that CROMEMCO 16K BASIC can determine the baud rate of your terminal.
2. CROMEMCO 16K BASIC will respond with a prompt {>>} to indicate that it is ready to accept a command.

CROMEMCO 16K BASIC FROM PAPER TAPE

Memory Requirements:

1. 16K of RAM starting at location 8ØØØH
2. at least 8K of RAM starting at location Ø

System Set Up:

For a CROMEMCO Z-1 or Z-2 computer with the RDOS monitor on a 4FDC board

1. set the power on jump to CØØØH (monitor)
2. set all of the 4FDC switches 'OFF'
3. modify the 'PAPER TAPE LOADER' instruction at 25H from JP EØØ8H to JP CØØØH

In all other cases use the CROMEMCO Z-8Ø monitor

1. set the power on jump to EØØØH (monitor)
2. use the 'PAPER TAPE LOADER' as provided

Method:

1. enter the 'PAPER TAPE LOADER' supplied with your CROMEMCO 16K BASIC at location Ø
   (type {SMØ} then enter the program)
2. mount the paper tape in the reader
3. start the program execution at location Ø {GØ}
4. turn on the paper tape reader

NOTE:
If you are using a teletype and the program exits to the monitor while the tape is being read in, this means a checksum error has been encountered. To correct:
   a. stop the paper tape
   b. back the paper tape up about 2 feet
   c. start the program execution at location Ø
   d. turn on the paper tape reader

(It is not necessary to start reading the program in from the beginning of the tape as loading address information is stored on the tape.)

5. after the CROMEMCO 16K BASIC program has been loaded, shut off the paper tape reader.
6. the 'PAPER TAPE LOADER' will return control to the monitor.
7. start program execution at location 8ØØØH {G8ØØØ}
8. depress the return key on the console several times so that CROMEMCO 16K BASIC can determine the baud rate of your terminal.
9. CROMEMCO 16K BASIC will respond with a prompt {>>} to indicate that it is ready to accept a command.

AUTOMATIC STARTUP AND PROGRAM EXECUTION FROM CDOS

A very powerful feature of the Cromemco Disk Operating System (CDOS) is the ability to enter directly into an application program when powering up the computer. This is especially useful for the inexperienced user as there is no need to deal with any of the commands which are used to load and execute a program.

If, for example, the user wants to execute the BASIC program 'START.SAV' automatically when CDOS is entered, the following steps should be followed:

1. Make sure that there is a copy of the batch command file '@.COM' on disk A.

2. Save the BASIC program you want RUN in a file (in this example we are using 'START.SAV'). The program must be SAVEd (not LISTed) in order for this to work!

Our program for this example is:

```
100 REM THIS IS MY APPLICATION PROGRAM!
110 A = 5
120 B = 10
130 PRINT "THE ANSWER IS: "; A*B
140 END
```

3. Using the editor, create a file named 'STARTUP.CMD' on disk A. Note that this must be named 'STARTUP.CMD' as this is the file name that CDOS looks for. In this example, the command file should contain the line:

BASIC START.SAV

Then, when CDOS is entered, the batch command will call BASIC which will RUN the saved program 'START.SAV'.

4. When the computer is turned on and CDOS is entered (you have to hit the carriage return several times), our example will output the following:

```
CROMEMCO CDOS VERSION ØØ.2Ø

A.@ STARTUP
BATCH  VERSION ØØ.Ø3

A.BASIC START.SAV

CDOS 16K BASIC, VERSION 5.Ø

THE ANSWER IS: 5Ø

***14Ø END***
```

ENTERING A NEW PROGRAM

When beginning a new program, it is advisable to type SCR, which is a scratch command, to clear the user area of any stray statement lines that may be present. Of course, when power is applied and the system is "brought up" for operation, no program is immediately active. However, typing SCR each time a new program is to be input is a good programming habit.

When typing a new program, the user must be certain to begin each line with a line number (which may be any integer from 1 to 99999) and end each line with a carriage return (CR). The lines may be entered out of numerical sequence, such as:

```
20  A = 2
30  B = 4
15  DIM A$ (50)
25  DIM B$ (50)
```

16K BASIC will rearrange these lines to:

```
15  DIM A$ (50)
20  A = 2
25  DIM B$ (50)
30  B = 4
```

Before running a program, the user should LIST the program on the terminal and check for obvious typing errors. Corrections to a program can be made by retyping lines containing errors, by deleting lines or by adding lines. A line containing errors may be rewritten by retyping the line number and the corrected text.

In the example above, if you wish to change the 4 in line 30 to an 8, you would type:

```
30  B = 8
```

When the program is listed, the computer will automatically replace the original line 30 with the corrected version. Thus, the program will be relisted as follows:

```
>>  LIST
15  DIM A$ (50)
20  A = 2
25  DIM B$ (50)
30  B = 8
```

Once a program has been written and edited, it can be executed with the RUN command.

```
>>10  PRINT "A PROGRAM"
>>20  PRINT "TO DEMONSTRATE"
>>30  PRINT "THE RUN INSTRUCTION"
>>RUN
A PROGRAM
TO DEMONSTRATE
THE RUN INSTRUCTION
```

To add lines to a program, you simply type a new statement with a line number which falls between the numbers of existing statements in the program. Consequently, it is good programming practice to increment by 10 when numbering statement lines in case statements must be inserted later. If the number of lines to be inserted between statements exceeds the range of available line numbers, the RENUMBER command may be used to reassign numbers.

The format for the RENUMBER instruction is:

```
>>RENUMBER X,Y
```

where X is the starting number for the lowest-numbered statement line and Y is the increment. For example, we might renumber the lines in the following program to start with 10 and increment each number by 10:

```
 5  A = B
 6  PRINT A
10  B = B*B
12  PRINT B
13  END
>>  RENUMBER 10, 10
```

The program would appear as follows:

```
10  A = B
20  PRINT A
30  B = B*B
40  PRINT B
50  END
```

To stop execution of listing of a program, or to interrupt any task being performed, press and release the ESCape key. The computer will revert to command mode.

When you have finished using 16K BASIC, you may pass control to the Cromemco Monitor of CDOS (or the Resident Operating System) or you may simply turn off the machine. Before

"logging off", be sure that any program you wish to keep is saved in "hard copy" either on paper tape, floppy disk, cassette tape, or some other permanent format. When power to the system RAM memory is removed, the memory contents are lost.

NOTE: It is important to remove disks from the disk drive before turning the power off or on. This will eliminate the possibility of stray bits being written on the disk as the system is powered up or down.

1.5 STEPPING THROUGH A BASIC PROGRAM

Like other computer languages, 16K BASIC has its own format, rules of grammar, and user protocol. The best way to learn BASIC is to sit down in front of a working computer system and try each command. This section presents a short program that will help introduce you to the BASIC language.

Once you have plugged in your terminal, switched the power on, and loaded BASIC (see Section 1.4), the computer should print out the BASIC prompt (>>).

The ">>" symbol typed at the left is a 'go ahead' signal from the computer. This signifies that the computer is ready to accept program commands. At this point, the user may begin inputting BASIC statements. Note that after each statement is input the user must hit carriage return (CR).

The program we will be inputting as an example is a short program which calculates the average of five numbers. We begin by stating a line number (see STATEMENT LINE NUMBER in the Glossary for more information about statement lines). You may enter line numbers individually by typing any number from 1 to 99999 after the >> marks.

For example, you may enter the following line:

```
>> 10 PRINT "BEGIN BASIC PROGRAM" (CR)
```

If you wish to let the computer take care of the line numbering after the 'ready' marks, type:

```
>>  AUTOL 10, 10      (CR)
```

and the computer will respond with:

```
>>  10
```

This is the first line number. Further line numbers will increment by 10. (To change the AUTOL format, see the AUTOL command definition.)

Next, type in your first instruction:

```
10  LET N = 5        (CR)
```

When the program is RUN, this instruction will assign a value of 5 to the variable N. BASIC will remember this value and it will remain the same until redefined by the user or by an algorithm within the program. BASIC will also understand the statement above without the LET, as in:

```
10 N = 5             (CR)
```

This is called an 'implied LET' and is a characteristic of advanced versions of BASIC. BASIC will also disregard spacing of characters in statement lines. Thus, for example, the statement:

```
10  LET N = 5        (CR)
```

is equivalent to the statement:

```
10  LET N=5          (CR)
```

At this point, we have the following lines on the terminal:

```
AUTOL 10, 10
10  N = 5
```

BASIC is waiting for our next instruction. Type in:

```
20  INPUT A: INPUT B: INPUT C: INPUT D: INPUT E    (CR)
```

With this long statement line, we have created a set of commands that will ask for five numbers (inputs A,B,C,D and E) when the program is actually run. In 16K BASIC, more than one statement line can be packed behind a line number. These statement lines must be separated by colons. Packing statement lines saves memory space.

Next, we enter the statement line:

```
30  X = (A+B+C+D+E)/N          (CR)
```

This statement line assigns to variable X the value of the sum of the five input numbers divided by N. The variable X is therefore the arithmetic mean, or average, of these five values. We defined N as 5 previously. If we do not define N, then it is assumed that N = 0.

Naturally, we'll want to look at X to see what it is. We type in:

```
40  PRINT X                    (CR)
```

which instructs BASIC to print the value of X when the program is run. What do we have so far? Our program, as listed below,

looks ready to run.

```
AUTOL 10, 10
10  N = 5
20  INPUT A: INPUT B: INPUT C: INPUT D: INPUT E
30  X = (A+B+C+D+E)/N
40  PRINT X
```

16K BASIC does not require an END or STOP statement line as the last statement line, but it is good programming practice to include one. So we can add the following statement to our program:

```
50  END                          (CR)
```

If we wanted to run through the program a number of times, finding means for many groups of five numbers, we might change line 50 to:

```
50  GOTO 10                      (CR)
```

This would instruct the computer to return to line 10 and re-execute the program, finding the mean of five new numbers each time. To escape from such a loop, or from a program during its execution, press the escape key on your terminal or teletype keyboard and release it.

Now let's run the program. After statement line 50 is typed in we hit the ESC (escape) key to get out of the autoline line numbering. Then we hit carriage return and type in:

```
RUN                              (CR)
```

This command instructs the computer to begin execution of the program. When the first INPUT command in line 20 is encountered during program execution, a question mark (called a prompt) will appear on your terminal, as indicated below:

```
RUN
?
```

This prompt indicates that the computer is waiting for you to input a number for the first variable in the INPUT statement. We therefore type in:

```
?17
```

and BASIC responds with:

```
?17
?
```

The second prompt indicates that the computer is waiting for a value for INPUT B. In fact, we need to enter values for the next four INPUT variables. Once we input these values and hit carriage return, the computer will complete program execution and print out the value of X, as shown below:

```
RUN
?17
?25
?12
?18
?13
17
***50 END***
```

Almost instantly, the BASIC program has calculated and printed the average of the five numbers we input. This average is equal to 17. At this point, we might want to do a little formatting to change the way in which the results are printed out. We LIST the program again to see what we have to work with.

```
LIST
10  N = 5
20  INPUT A:INPUT B:INPUT C:INPUT D:INPUT E
30  X = (A+B+C+D+E)/N
40  PRINT X
50  END
```

We can start by inserting some statement lines within the present text of the program. At any time, lines can be inserted between present statement lines by using a number which falls between two consecutive statement line numbers. For example, a line numbered 15 will be inserted between statement number 10 and statement number 20. When a program is RUN or LISTed, the new statement lines are automatically inserted into the proper space in the hierarchy.

We decide to add spaces between lines to make reading easier. We add:

```
15  PRINT
16  PRINT
35  PRINT
36  PRINT
```

Inserting a PRINT instruction in a statement line without text to be printed generates a linefeed. To save time, you may use an @ symbol in a statement line instead of PRINT. At this point, you'll want another look at the program. BASIC will have sandwiched the new statement lines into the old text as shown below:

```
10  N = 5
15  PRINT
16  PRINT
20  INPUT A:INPUT B:INPUT C:INPUT D:INPUT E
30  X = (A+B+C+D+E)/N
35  PRINT
36  PRINT
40  PRINT X
50  END
```

Now type RUN. According to our program, the computer will skip two lines after you type in RUN and then wait for you to type in 5 numbers. Once these numbers are input, the program instructs the computer to skip two more lines and then type the result. This output appears on the terminal as follows:

```
RUN

?22
?14
?16
?20
?28

20
***50 END***
```

Many more variations may be added. For example, the statement line:

```
5 PRINT "A PROGRAM TO CALCULATE THE AVERAGE OF 5 NUMBERS"
```

will give you a title for this program. You might also replace line 40 with:

```
40 PRINT "THE MEAN OF THESE NUMBERS IS ";X
```

To rewrite any line, just retype the line number and the statement text. The rewritten line will automatically replace the old line. Variables do not need to be within quotation marks to print.

Endless time could be spent examining all options, but the best way to learn BASIC is to try out everything that catches your interest. Several hours of practice with the BASIC instructions will give you a solid grounding in BASIC. Refer to Chapters 2 and 3 for a description of the statements, commands, and functions available in Cromemco 16K BASIC.

2.0 16K BASIC PROGRAM INSTRUCTIONS

Brief descriptions of each of the program statements and commands included in Cromemco 16K BASIC are presented in the following sections. These descriptions cover the purpose or use of each instruction, a general format for each, and specific examples.

All program instructions described in Sections 2.1 and 2.2 must include a line number if they are to be used as statements. Although these line numbers are not included in the general form of each instruction described in these sections, it is assumed that the user will precede each statement with a number.

All instructions described in Sections 2.1, 2.4, 2.5, and 2.6 may be used as statements (preceeded by a line number, for execution at run time) or commands (used without a line number, for immediate execution--useful for debugging and testing).

All instructions described in Section 2.2 can only be used as statements.

All instructions described in Section 2.3 can only be used as commands.

MULTIPLE STATEMENTS

STATEMENT : STATEMENT : STATEMENT

Cromemco 16K BASIC allows multiple statements per line with a few exceptions and restrictions. The number of statements which may be placed on a line is limited only by the length of the line. User defined functions (DEF FNs) and DATA statements must be on a line by themselves. The statements and commands listed below may be part of a multistatement line but they must be the LAST statement on the line, i.e., no other statement may follow on the same line.

RUN	REM
GOTO	GOSUB
FOR	ON
ENTER	DELETE

See the IF - THEN command for an example.

2.1 INSTRUCTIONS WHICH CAN BE USED AS BOTH STATEMENTS AND COMMANDS

DEG

This command sets the DEGrees mode for trigonometric calculations.

The general format for this command is simply:

DEG

DELETE

The DELETE command is used to remove statement lines from a program. The general format for this command is:

DELETE n_1, n_2

where n_1 and n_2 are line numbers in the current program. The DELETE command removes all statement lines between and including any two specified line numbers (n_1 and n_2).

Example:

```
10  INPUT A, B, C
20  D = A+B+C
30  PRINT A
40  PRINT B;C
50  PRINT D
60  END
DELETE 30, 40
LIST
10  INPUT A,B,C
20  D = A+B+C
50  PRINT D
60  END
```

In this example, the effect of the DELETE command is to remove lines 30 and 40 from the program.

DIM

The DIM statement is used to define the size of a matrix (see the definition of Matrix in Section 1.2) or a string variable. Cromemco BASIC permits the user to define one, two, or three dimensional matrices. A two (i.e., M_{ij}) or three (i.e. M_{ijk}) dimensional matrix is commonly called a table or array. A one dimensional matrix -- a matrix with n columns but only one row -- is commonly called a list or vector. When dimensioning a string variable, the computer simply handles the variable as though it were a one dimensional array.

The respective general formats for declaring the number of rows and columns in one, two, and three dimensional matrices are:

```
DIM M(n) or M$(n)
DIM M(n,m)
DIM M(n,m,r)
```

where M is any matrix variable, M$ is any string variable, and n, m, and r are integers. (A matrix variable can have the same name as any scalar.)

If a matrix is not specifically dimensioned in a program, the default value of 10 will be automatically assigned to a singly subscripted matrix. Doubly and triply subscripted matrices will generate an error message if not explicitly dimensioned.

The maximum size of any matrix is only restricted by the amount of available memory.

FOR...NEXT

The FOR...NEXT statements instruct the computer to repeat a group of statements a specified number of times. Once this "loop" has been executed for the last time specified, program control is transferred to the statement line immediately following the NEXT statement.

The general format for this command is:

```
FOR X = n1 to n2 STEP n3
    :
  (n program statements)
    :
NEXT X
```

The X in a FOR-NEXT command may be any non-subscripted numeric variable. String variables cannot be used in FOR-NEXT commands. The expressions n_1 and n_2 may equal to <u>any</u> number. The STEP n_3 portion of the FOR command is optional. If a STEP value is not specified, the computer assumes a default value of +1 for STEP. Note that 16K BASIC programs will execute significantly faster if variables used in loops (and/or for subscripting) are declared as type INTEGER.

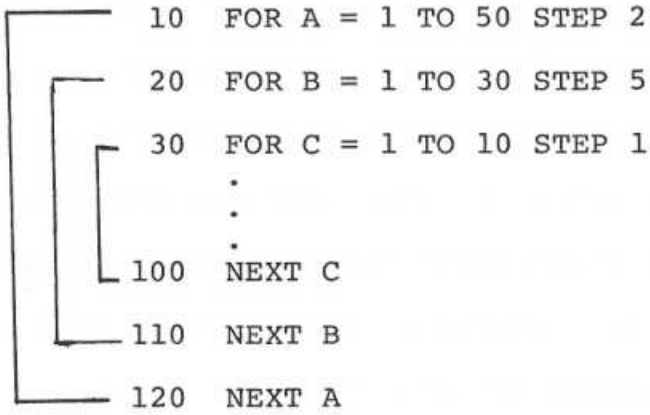
Example:

```
10  FOR A = 0 to 10 STEP 2
20  B = A+1
30  PRINT B;
40  NEXT A
50  END
RUN
1357911***50END***
```

Any FOR-NEXT loop is executed by first setting the value of the FOR variable equal to n_1. After all the statements in the loop are executed, the program returns to the FOR command and increments the FOR variable by the value specified by n_3 (the STEP value). The loop is terminated when the value of the FOR variable is greater than n_2 and control is transferred to the statement immediately following the NEXT statement.

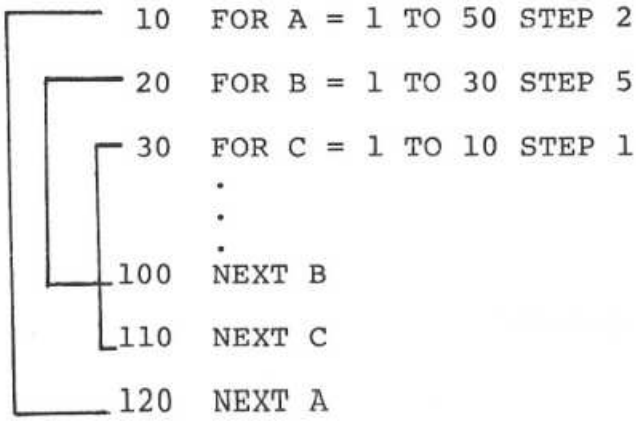
FOR-NEXT loops may be nested within a program. It is important to keep in mind, however, that each FOR command and its corresponding NEXT command must be completely contained within any larger loop.

Example: Correct Nesting Format



Illegal nesting occurs when FOR-NEXT loops overlap.

Example: Incorrect Nesting Format



GOTO

The GOTO instruction is used to interrupt the normal execution sequence of statements in a program by transferring control to a specific line.

The general form of the GOTO statement is:

```
GOTO n
```

where n is the number of a statement in the program. GOTO can also be used in direct mode and can be used after you hit RUN.

This instruction is particularly useful for transferring program control to a subroutine if a certain logical condition is met.

Example:

```
10  INPUT A
20  IF A<0 THEN 200
30  B = SQR(A)
40  PRINT "THE SQUARE ROOT OF ";A;" IS ";B
50  GOTO 210
200 PRINT "THIS YIELDS AN IMAGINARY NUMBER"
210 END
RUN
? - 2
THIS YIELDS AN IMAGINARY NUMBER
***210 END***
RUN
?3
THE SQUARE ROOT OF 3 IS 1.7320508075688
***210 END***
```

In this example, the GOTO statement transfers program control to line 210.

GOSUB-RETURN

The GOSUB command transfers control to the first statement of a subroutine.

The general format for the GOSUB command is:

```
50  GOSUB n
```

where n may be the starting line number of any subroutine. An arithmetic expression may be used if it evaluates to an integer that is a legal line number. GOSUB can also be used in direct mode and can be used after you hit RUN.

When the desired subroutine has been performed, a RETURN command instructs the computer to exit the subroutine and to re-route program control back to the statement line immediately following the one containing the GOSUB command.

Example:

```
20  GOSUB 50
30  PRINT "PROGRAM OVER"
40  END
50  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***
```

In 16K BASIC, subroutines may be fully enclosed or "nested" in another subroutine. Various versions of BASIC will have different limits of nesting depth. 16K BASIC allows nesting to 16 deep.

When nesting subroutines, remember that the RETURN instruction will take you back to the last GOSUB and start execution of the next line immediately after. Improperly nested GOSUB-RETURN instructions will generate error messages. A nested subroutine is executed after the GOSUB statement and before the RETURN statement in the subroutine in which it is enclosed.

IF...THEN

The IF-THEN statement instructs the computer to transfer control to a specific statement in the program based on whether a specified condition is true or false.

The general form of the IF-THEN command is:

```
IF X THEN line-number
```

or

```
IF X THEN statement
```

where X may be any relational expression or numeric expression. In a relational expression, an operator (e.g., =, <,>=,<=, <>,>) is used to compare two expressions or values. The specified statement following THEN may be any legal BASIC statement with the exception of a FOR-NEXT, END, REM or DATA statement.

Example:

```
70  IF A = B/C THEN 100
80  IF A < B/C THEN 110
90  IF A > B/C THEN PRINT "NO REAL NUMBER"
```

In this command, the expression following IF is evaluated and if it is true the statement following THEN is executed along with any subsequent statements on the same line. If the expression is false, the program continues to the LINE immediately following the IF-THEN command.

Example:

```
1ØØ INPUT A : IF A=Ø THEN PRINT "MUST BE NON-ZERO" : GOTO 1ØØ
11Ø PRINT A
12Ø END
```

In this example, the computer will output a prompt (?) and the user will respond with a number. If the number is non-zero (A=0 is false), control will be passed to line 11Ø, the number will be printed, and execution of the program will terminate. If the number is zero (A=0 is true), the part of line 1ØØ after THEN will be executed, printing out the message and returning control to the beginning of line 1ØØ which will request another number from the user.

IMODE

To set all variables within a program to the Integer mode the instruction:

```
IMODE
```

should be given as a command before the program is RUN or as the first statement in the program. To make sure that the BASIC interpreter is in the Integer mode, the line

```
1 IMODE: X=2.5: IF X=2.5 THEN RUN
```

may be coded. If this line is used, it must be the first line of the program.

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

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

INPUT

The purpose of this statement is to assign a value, input through a terminal, to a variable in a program. The general format for this instruction is:

INPUT X_1, X_2,...,X_n

where X_1 through X_n may be any numeric or string variables. When the INPUT statement is executed in a program, a prompt is output to the terminal. The user should respond to this prompt by typing in a list of data which corresponds to the variable list in the INPUT command.

Example:

```
10  INPUT A,B,C,D
20  PRINT A
30  INPUT E,F
40  PRINT B;" ";C;" ";D;" ";E;" ";F
50  END
RUN
? 32,18,20,4
32
? 100, 200
18 20 4 100 200
***50 END***
```

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.

Example:

```
INPUT A,B,C,D,E
RUN
?1,2,3      (CR)
??4
??5
```

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

The type of data input must be the same as the type of variable listed in the INPUT command. For example, if the INPUT command contains a list of numeric variables, the data input must be numeric data. If the data and variable types don't match, an error message will be generated.

INTEGER

To set a given variable to the integer mode the instruction:

INTEGER A [(X)] [,B(Y),...]

should be put at the beginning of the program. In this statement, X and Y are optional DIMensions (DIMensioning may be done via the INTEGER statement) and A, B, etc., are the INTEGER variables.

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

This instruction overrides the SFMODE and LFMODE instructions.

LET

The LET statement is used to assign a value to a given variable, string variable, or array. The general form of the LET statement is:

LET X = n

where X is any numeric or string variable and n may be a number, another variable, or the results obtained by evaluating an expression.

16K BASIC is designed to allow the user to assign values to variables without entering LET each time. This capability is called "implied LET".

Examples:

LET STATEMENTS	IMPLIED LET STATEMENTS
10 LET A = 45	10 A = 45
10 LET B = A + 10	10 B = A + 10
10 LET A$ = "16K BASIC"	10 A$ = "16K BASIC"
10 LET X = Y + L	10 X = Y + L

LFMODE

To set all variables within a program to the Long Floating Point mode the instruction:

```
LFMODE
```

should be given as a command before the program is RUN or as the first statement in the program. To make sure that the BASIC interpreter is in the Long Floating Point mode, the line

```
1 LFMODE:  X=0.12345678: IF X<>0.12345678 THEN RUN
```

may be coded. If this line is used, it must be the first line of the program.

Long Floating Point variables occupy 8 bytes, have an accuracy of 14 digits, and must be within the range $\pm$9.99E+62 to $\pm$9.99E-65.

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

LIST

The LIST command instructs the computer to print out one or more statement lines to an output device or a file. The most frequently used output devices include terminals, line printers, and disk sectors.

The LIST command may be used to output an entire program, a block of statement lines within a program, or a single statement line.

The general format for this command is:

LIST n_1,n_2

where n_1 and n_2 are statement numbers in the current program.

When n_1 and n_2 are not specified, the LIST command will output the entire program. If a single statement number is specified, the program will be listed from that statement number through the end of the program. If both n_1 and n_2 are specified, the computer will list line number n_1 through line number n_2.

Example:

```
10  INPUT A, B
20  X = A+1
30  PRINT A, B
40  PRINT X
50  END
```

```
LIST 20, 30
20  X = A+1
30  PRINT A, B
LIST 30
30  PRINT A, B
40  PRINT X
50  END
LIST
10  INPUT A, B
20  X = A+1
30  PRINT A, B
40  PRINT X
50  END
```

When the file version of the LIST command is used (see Chapter 3), the current program is automatically output to a file device in ASCII format. A program output by the LIST command can be read back into storage by the ENTER command.

LONG

To set a given variable to the Long Floating Point mode the instruction:

LONG A [(X)] [,B(Y),...]

should be put at the beginning of the program. In this statement, X and Y are optional DIMensions (DIMensioning may be done via the LONG statement) and A, B, etc., are the Long Floating Point variables.

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

This instruction overrides the IMODE and SFMODE instructions.

ON...GOTO

ON...GOSUB

These commands transfer control to any one of several lines in a program based on the value of the expression contained in the statement.

The general form for these statements is:

ON (X) GOTO or GOSUB $n_1, n_2, \ldots, n_i$

where X may be any arithmetic expression which is evaluated to an integer and n_i is any statement line number within the program.

Example:

```
 10  INPUT A
 20  LET A = A + 1
 30  ON A GOTO 60, 80, 100, 120
 60  PRINT A; " ITERATIONS REQUIRED"
 70  GOTO 20
 80  PRINT A; " ITERATIONS REQUIRED"
 90  GOTO 20
100  PRINT A; " ITERATIONS REQUIRED"
110  GOTO 20
120  END
RUN
?0
1  ITERATIONS REQUIRED
2  ITERATIONS REQUIRED
3  ITERATIONS REQUIRED
***120 END***
```

In both command statements, control is transferred to the line number whose position in the list corresponds to the value of the expression. Thus, if the value of the expression equals 2, control is transferred to the second statement in the list. If the value of an expression equals a real number, the program ignores the fractional part of the number and treats the number as an integer. If the value of the expression is less than or equal to Ø or greater than the number of items in the list, the ON-GOTO or ON-GOSUB command is ignored and the program continues to the next statement.

Any statement line listed in an ON-GOSUB command must be the first line of a subroutine in the program.

PRINT

The PRINT statement is used to print out expressions or strings on a terminal or teletype. The PRINT command can also be used to skip a line.

The general format of the PRINT statement is:

```
PRINT X
```

where X may be a number, a variable, a string variable, or a string literal. The characters in a string will be printed exactly as they appear. When used alone, the PRINT command will generate a linefeed.

The following program contains a number of examples of the use of the PRINT statement.

Example:

```
 10  A = 1977
 20  REM  B = YOUR BIRTHDATE
 30  PRINT "THIS IS ";A
 40  PRINT
 50  PRINT
 60  INPUT B
 70  C = A-B
 80  PRINT "YOU WERE BORN IN ";B
 90  PRINT
100  PRINT "YOU ARE "; C; " YEARS OLD THIS YEAR"
110  END
```

```
RUN
THIS IS 1977
?1952
YOU WERE BORN IN 1952
YOU ARE 25 YEARS OLD THIS YEAR
***110 END***
```

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

Items separated by commas are printed beginning in the leftmost column of each printing field. Using the default value of 18 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    Ø         18        36        54
 no.
```

See the SET instruction to change the default value of 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"
RUN
```

will produce output in the following format:

```
ALTOGETHER
```

Commas and semicolons can be used in a PRINT statement in any combination. If more items are listed in a PRINT statement than can be output on one line, the computer will generate a linefeed and continue printing on the next line.

The PRINT command may be replaced by the symbol "@" in 16K BASIC. For example, the following statements:

```
30  PRINT
40  PRINT "AND IT SAVES SPACE"
```

can also be written as:

```
30  @
40  @ "AND IT SAVES SPACE"
```

On some teletypewriters and video displays, this facilitates entry of large, text-oriented programs with many blank lines.

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

RAD

This command sets the RADians mode for trigonometric calculations. SCR and RUN automatically set the RAD mode.

The general format for this command is simply:

RAD

(See SIN, COS, TAN, and ATN functions for examples.)

RANDOMIZE

This statement is used to reset the random number dummy variable used by the RND and IRN functions so as to produce a different sequence of random numbers each time the RND and IRN functions are run.

Example:

```
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 the program is executed.

RANDOMIZE should be used sparingly (or even only once) within a program to ensure that a truly random sequence of numbers results.

READ

The READ statement is used to read values from a DATA list and assign these values to the variables listed in the READ command.

The general form of this statement is:

READ X_1, X_2,...,X_n

where X_1 through X_n may be any numeric or string variables.

The order in which variables appear in the READ statement determines which value from the DATA list will be assigned to that 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. 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 too few DATA elements are input, an error message will result.

Example:

```
10  READ A,B,C,D$
20  PRINT A,B,C,D$
30  DATA 10,20,30,"END"
40  END
RUN
10              20              30              END
***40 END***
```

Note that the DATA elements corresponding to a numeric variable must be numeric data and the element corresponding to a string variable must be a string.

REM

The REM statement is used to insert remarks or comments in a program. The general format for this statement is:

```
REM text
```

where text can be any comment or series of characters.

REM statements included in a BASIC program are ignored when the program is executed but are output exactly as entered. Consequently, 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. 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.

RESTORE

The RESTORE statement allows rereading of data by resetting the pointer in the DATA list to the first data item.

The general format for this command is:

RESTORE line number

where the line number is any line number in the current program. This command instructs the computer to reset the pointer to the first data element in the first DATA statement at or after the line number stated after RESTORE. If no line number is stated, the pointer is reset to the first data element in the program.

Example:

```
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***
```

RUN

The RUN command instructs the computer to execute the program starting at the lowest numbered line.

The general format for this command is:

```
RUN
```

Example:

```
10  A = 5
20  PRINT A
30  B = 16
40  PRINT B
50  END
RUN
5
16
***50 END***
```

In this example, the RUN command instructs the computer to execute the entire program.

Use of the RUN command automatically resets or "clears" all variables, string variables, and matrices.

SCRatch

The SCR command deletes the current program from memory. The programmer should keep in mind that the SCR command erases everything in the user work space and that scratched programs cannot be recovered. Once the work space has been cleared, the user may input a new program or access a previously input program which has been saved on either disk or some storage device.

The general format for the SCRatch commands is simply:

```
SCR
```

Example:

```
10  X=4
20  INPUT Y
30  Z = X*2 + Y
40  PRINT Z
50  END
SCR
```

In the above example, statement lines 10 through 50 are deleted from memory and the user can input a new program.

SFMODE

To set all variables within a program to the Short Floating Point mode the instruction:

SFMODE

should be given as a command before the program is RUN or as the first statement in the program. To make sure that the BASIC interpreter is in the Short Floating Point mode, the line

1 SFMODE: X=0.12345678: IF X=0.12345678 THEN RUN

may be coded. If this line is used, it must be the first line of the program.

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.

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

SHORT

To set a given variable to the Short Floating Point mode the instruction:

SHORT A[(X)][,B(Y),...]

should be put at the beginning of the program. In this statement, X and Y are optional DIMensions (DIMensioning may be done via the SHORT statement) and A, B, etc., are the SHORT Floating Point variables.

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.

This instruction overrides the IMODE and LFMODE instructions.

2.2 PROGRAM INSTRUCTIONS WHICH CAN ONLY BE USED AS STATEMENTS

DATA

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

The general form for this statement is:

DATA n_1, n_2,...,n_i

where n is any numeric value or any string and i is an integer. Values included in a DATA list are separated by commas. More than one DATA statement may be included in any program. Data is read in order from the first to the last DATA statement appearing in a program and from left to right within the statement. String data must be enclosed in quotation marks.

Example:

```
10  READ A,B,C,D$
20  PRINT A,B,C,D$
30  DATA 16.8, 20.4, 76.2
40  DATA "END"
50  END
RUN
16.8            20.4            76.2            END
***50 END***
```

If the computer runs out of DATA during program execution, an error message will be printed.

END

The END statement is comparable to STOP in that it halts program execution and causes the computer to return to command mode. The only real difference is that a program may not be continued after an END statement. The general format for this statement is simply:

```
END
```

Example:

```
10  LET A = 100
20  PRINT A; SPC(1); A+1; SPC(1); A+2; SPC(1); A+3
30  END
RUN
100 101 102 103
***30 END***
```

In Cromemco 16K BASIC, END statements are allowed anywhere in a program. Also, END statements are not required at the end of a program. However, it is considered good programming practice to use the END statement as the last statement in a program.

STOP

The STOP statement halts program execution and causes the computer to enter command mode. Program execution may be restarted from the beginning of the program or by using the CONtinue command from the statement line immediately following the last line executed. The general format for this statement is simply:

STOP

Example:

```
10  INPUT A,B,C,D,E,F
20  LET N=A+B/C
30  PRINT A;B;C'
40  STOP
50  PRINT D,E,F
60  PRINT N
70  END
RUN
?1,2,3,4,5,6
123
***40 STOP***
>>
```

In this example, the computer goes back to command mode after encountering the STOP in statement line 40.

2.3 PROGRAM INSTRUCTIONS WHICH CAN ONLY BE USED AS COMMANDS

AUTOL

The AUTOL command provides automatic statement line numbering. The general format for this command is:

```
AUTOL n, m
```

where n is the number of the first statement in the current program and m is an integer which specifies the amount to increment between line numbers. The AUTOL command generates line numbers automatically so that the user does not have to assign a line number to each statement in the current program as the statement is input.

To interrupt the automatic generation of line numbers, press the ESCAPE key or the RETURN key when prompted for the next line.

CONtinue

The CON command instructs the computer to continue program execution after a STOP statement or to resume execution after a program is interrupted by a program error or by the user pressing the ESCape key. The general format for the CONtinue command is simply:

```
CON
```

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

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

DIR

The DIR command corresponds to the CDOS DIR command (see the CDOS User's Manual for a full description). The DIR command lists disk files giving size (in K-bytes) and number of extents. However, in BASIC, the file specifier (if used) must be enclosed in quotation marks or must be another valid form of string expression.

Examples:

DIR -- will list all files on the current disk.

DIR "A:*.*" -- will list all files on drive A.

DIR "*.SAV" -- will list all files on the current disk with the entension "SAV".

RENUMBER

This command renumbers the statement lines in the current program. The general format for the command is:

```
RENUMBER  m, n, b, e
```

where m is the starting line number in the RENUMBERed program, n is an integer which specifies the amount to increment between line numbers, and b and e are the first and last existing line numbers to be RENUMBERed.

The RENUMBER command also alters line numbers imbedded in the program in GOTO, GOSUB, and IF-THEN statements to conform to the renumbered statements.

The default value for the RENUMBER statement is a starting line number (m) of 10 and an increment value (n) of 10.

Example:

```
  11 INPUT A
  24 INPUT B
  37 PRINT A*B
  50 GOTO 11
  63 END

>>RENUMBER

>>LIST

  10 INPUT A
  20 INPUT B
  30 PRINT A*B
  40 GOTO 10
  50 END
```

If only a starting line number (m) is specified, the RENUMBERed program will start with line number m and also be incremented by m.

Example:

```
>>RENUMBER 100

>>LIST

  100 INPUT A
  200 INPUT B
  300 PRINT A*B
  400 GOTO 100
  500 END
```

If both of the first two parameters (m, n) are specified, the RENUMBERed program will start with line number m and will be incremented by n.

Example:

```
>>RENUMBER 100,10

>>LIST

  100 INPUT A
  110 INPUT B
  120 PRINT A*B
  130 GOTO 100
  140 END
```

When the third parameter (b) is specified, only the line numbers in the old program starting with b will be altered in accordance with m and n.

Example:

```
>>RENUMBER 1000,150,120

>>LIST

  100 INPUT A
  110 INPUT B
 1000 PRINT A*B
 1150 GOTO 100
 1300 END
```

If all four parameters are specified, line numbers in the old program between b and e will be changed.

Example:

```
>>RENUMBER 1000,1,110,1150

>>LIST

  100 INPUT A
 1000 INPUT B
 1001 PRINT A*B
 1002 GOTO 100
 1300 END
```

2.4 STRING CONTROL INSTRUCTIONS

STRING PROTOCOL

Cromemco 16K BASIC includes a number of procedures for manipulating alphabetic information. In BASIC, alphabetic information is handled through strings and string variables. A string is defined as any combination of characters, including letters, numbers, special characters, and spaces, but excluding quotation marks. A string literal is defined as any string enclosed in quotation marks. For example, in the following statement:

```
10  PRINT "CROMEMCO 16K EXTENDED BASIC"
```

the phrase CROMEMCO 16K EXTENDED BASIC is a string literal. The "value" of a string is simply the number and sequence of characters which comprise the string literal.

String variables are also permitted in Cromemco 16K BASIC. A string variable is defined as any letter A through Z followed by a dollar sign or any letter and any number Ø through 9 followed by a dollar sign. Examples of legal string variables include the following:

String Variables

A$

B$

C5$

D6$

There is 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 16K BASIC for string size is 11 characters or less. If string literals of more than 11 characters are to be assigned to a variable, the string variable must be DIMensioned. The DIM statement is typically used in BASIC to define the size of an array (see the description of DIM in Section 2.1). When dimensioning a string variable, the computer simply handles the variable as though it were a one-dimensional array.

Example:

```
10  DIM A1$ (20), B$ (30), C4$(40)
```

In this example, the string variable A1$ is dimensioned to allow for strings up to 21 characters in length, the variable B$ is dimensioned to allow for strings up to 31 characters in length, and the variable C4$ is dimensioned to allow for strings of up to 41 characters in length. Any string literal 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 literal to be input.

Note: DIM A$ (20) allows a 21 character string because string bytes are numbered from 0 through the specified DIM size.

Remember that using the Øth element of strings (and arrays) can save memory space.

Strings may also be used in relational expressions (i.e., equal to, less than, greater than, etc.). For instance, two strings are equal if they have the same number and sequence of characters.

Example:

```
10  DIM A$ (20)
20  A$="MARY K. SMITH"
30  B$="MARY"
40  C$=A$(Ø,3)
50  PRINT B$
60  PRINT C$
70  END
RUN
MARY
MARY
***70 END***
```

In this example, B$ is said to be equal to C$.

Other string relations are determined by comparing in sequence the characters in the same position in both strings. The comparison is made on the basis of the ASCII code value assigned to each character. IF the ASCII code value of a character in a particular position in one string is greater than the value of the character in the same position in the other string, then the first string is greater than the second string.

A string value may be assigned to a string variable either through the assignment statement, the INPUT statement, or by using the READ statement and inputting string literal DATA. String data listed in a DATA statement must be enclosed in quotation marks.

Numeric and string variables may be mixed in the same READ statement. However, the data in the DATA list must match the variables in the READ statement. In other words, only numeric data can be assigned to a numeric variable and only string data can be assigned to a string variable.

Example:

```
10  DIM A$(20), B1$(30), C$(40)
20  READ A$, B1$, C, Y, C$, Z
30  DATA"THIS","EXAMPLE", 16.2,25.84,"ENDS",14
40  PRINT A$, B1$, C, Y, C$, Z
50  END
```

In this example, we read in two string variables first, two numeric variables next, another string variable, and another numeric variable. Consequently, the DATA list must first include two string literals, then two numeric values, then another string literal, and finally a numeric value.

Finally, the following example indicates an easy way to fill a string with any character (in this case, a question mark):

```
A$(-1)= "?" + A$(-1)
```

SUBSCRIPTING STRING VARIABLES

String variables may be subscripted in 16K BASIC. This capability allows the user to define and manipulate substrings. A substring is any part of a string. For example, if the string variable A$ equals "SUBSTRING EXAMPLE", substrings of A$ include "SUB","UBSTR", "G EXAMPLE", and any other parts of the string.

Substrings are referenced through subscripted string variables. The general formats for string subscripts are:

A$(n) or A$(n,m)

where A$ is any string variable and n and m are numbers or numeric expressions representing a character position within a string. Note that programs execute significantly faster if variables used for subscripting (and/or loops) are declared as type INTEGER.

The first format, A$(n), defines a substring starting with the character in position n and including all subsequent characters in the string. The second format, A$(n,m), defines a substring starting with the character in position n and ending with the character in position m.

Example:

```
10  DIM A$ (20)
20  LET A$ = "SUBSTRING EXAMPLE"
30  PRINT A$ (10)
40  PRINT A$ (3,8)
50  END
RUN
EXAMPLE
STRING
***50 END***
```

Remember that the first character position is numbered Ø!

A negative m specifies the length of a string. To specify a zero length, the user may code the second subscript as %8ØØØ% (-32768), because BASIC complements the user value to get the length (and the complement of %8000% is still %8000%) ignoring the most significant bit of the result.

This feature may be used in practice as follows:

```
1030  L=INT(J+L*(N-1))+K
1040  A$=B$+C$(M,BINOR(-L,%8000%))+D$
```

Thus, if L evaluated to zero in line 1030, statement 1040 would be equivalent to A$=B$+D$ since zero characters of C$ are specified! Note that, according to the rules of subscripts, C$(N,Ø) implies the portion of C$ starting at the Nth character and ending at the last character dimensioned!

The following conventions affect subscripting:

For A$ (N, M);

If N<Ø, then N is automatically set to equal Ø and M is set equal to the value specified in the DIM statement and any other specified value for M is ignored.

If M<Ø, then the length of the substring is set equal to the absolute value (ABS) of M.

If M<N, then M is set to the value specified in the DIM statement.

If N>DIM value, an error message is generated.

For output usage, if only one subscript (i.e., N) is specified, the string will be output from position N through the length of the string. If no subscript is specified, the string will be output from position Ø through the length of the string.

For input usage (i.e., READ, GET, LET, and INPUT), if no subscript is specified, the entire destination string will be set to nuls and then the source will be moved into it. If a single subscript N is specified, the string will be set to nuls from position N on and then the source moved into it starting at position N.

No string assignment will ever move more bytes to the destination than can be accepted by the specified dimension. (The user should note that A$ = A$ + B$ is not an effective construct as it is equivalent to A$ = B$. The user should specify A$(LEN(A$)) = B$.)

There are a number of BASIC statements and commands which can be used to manipulate strings. Most of these statements and commands have already been described in Section 2.1. However, BASIC also includes a number of functions designed to increase string handling capabilities. These pre-defined functions are described in Section 2.4.

2.5 PROGRAMMED FUNCTIONS

Cromemco 16K Extended BASIC includes a number of arithmetic and trigonometric functions which perform common, frequently used calculations. These functions are pre-defined in BASIC so that the programmer does not need to write a program to perform a common function everytime such a calculation is required.

Cromemco BASIC also includes a number of functions designed to increase string handling and matrix handling capabilities, a number of system functions which provide general system information, and an assembly language subroutine function.

In addition to these pre-defined functions, Cromemco BASIC permits the programmer to define up to 26 functions.

Cromemco 16K BASIC includes the following functions:

ARITHMETIC FUNCTIONS

ABS (X) - absolute value of X
BINAND(X,Y) - binary logical AND
BINOR(X,Y) - binary logical OR
BINXOR(X,Y) - binary logical EXCLUSIVE OR
EXP (X) - the value "e" to the power X
FRA (X) - gives only the fractional portion of X
FRE (X) - gives the number of memory bytes in the system which are currently "free" or unused
INT (X) - integer value of X
IRN (X) - generates an integer random number between 0 and 32767
LOG (X) - natural logarithm of X

MAX (X_1,...,X_n) - returns the numeric expression X_n with the maximum value in the expression list.

MIN (X_1,...,X_n) - returns the numeric expression X_n with the minimum value in the expression list.

RND (X) - generates a random number between 0 and 1

SGN (X) - algebraic sign of X

SQR (X) - square root of X

TRIGONOMETRIC FUNCTIONS

ATN (X) - arctangent of X (result in radians or degrees)

COS (X) - cosine of X

SIN (X) - sine of X

TAN (X) - tangent of X

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

LEN (X$) - gives the number of characters assigned to X$

POS (X$, Y$, n) - determines the location of a substring within a string by returning the value of the position at which the first character of the substring is located starting with character n.

STR$ (X) - converts any numeric expression X to a string which is the character representation of X

VAL (X$) - converts any string expression X$ to the numeric representation of X$

PROGRAMMER DEFINED FUNCTIONS

DEF FNs (X_1,...,X_n) = Y - allows the user to define up to 26 different functions

Brief descriptions, including examples, of each of these functions are included in the following section.

ARITHMETIC FUNCTIONS

ABS (X)

This function gives the absolute (i.e., positive) value of X, which can be any numeric expression.

Example:

```
10  PRINT ABS (-26)
20  END
RUN
26
***20 END***
```

BINAND, BINOR, BINXOR

These functions perform logical operations bit by bit on 16-bit operands. The BINAND function returns a 1 bit in a given position if both bits are equal to 1 and a Ø if both bits are not equal to 1. The BINOR function returns a 1 if either bit is equal to 1 and a Ø if both bits are equal to Ø. The BINXOR returns the exclusive or of each pair of bits.

The general form for these functions is:

```
BINAND (X,Y)
BINOR  (X,Y)
BINXOR (X,Y)
```

where X and Y are any numerical expressions (which are converted, if necessary, to 16-bit integers before the logical operation takes place).

Examples:

```
A = BINAND (30,%4F3%) will place 18 (%12%) in A
PRINT BINXOR (%32%,14) will print 60 (%3C%)
```

EXP (X)

This function calculates the value of the constant "e" (where e = 2.71828...) raised to the Xth power, where X is a numeric expression.

If X=12, for instance, the value calculated for EXP (12) is e^{12}.

Example:

```
10  PRINT
20  A = 4.1
30  B = EXP (A)
40  PRINT B
50  END
RUN
60.340287597344
***50 END***
```

FRA (X)

This function returns only the fractional portion of any numeric expression X. The integer portion of the number is removed and is not recoverable.

Example:

```
10  INPUT A
20  B = 3.7*A
30  PRINT FRA(B)
40  END
RUN
?  1
0.7
***40 END***
```

FRE (X)

This function gives the number of memory bytes in the system which are currently "free" or unused.

Example:

```
10  PRINT FRE(X)
20  END
RUN
2074
***20 END***
```

INT (X)

This function returns the largest integer value which is less than or equal to any numeric expression X.

The counterpart of this function is the FRA (X) function which ignores the integer portion of a given expression and returns only the decimal portion.

Example:

```
10  PRINT "ENTER NUMBER OF DAYS"
20  INPUT D
30  W = D/7
40  W = INT (W)
50  PRINT
60  PRINT "THAT'S ROUGHLY "; W; " WEEKS"
70  END
RUN
ENTER NUMBER OF DAYS
?36
THAT'S ROUGHLY 5 WEEKS
***70 END***
```

IRN (X)

This function generates an integer random number between Ø and +32767. The numeric expression X is really a dummy expression which is required but which is not actually used in the generation of the random number. To change this sequence, a RANDOMIZE command should be included in the program.

Example:

```
10  FOR N = 1 TO 10
20  PRINT IRN (6)
30  NEXT N
40  END
RUN
29284
25801
18835
 4647
 9295
18846
 4924
10105
20210
 7652
***40 END***
```

The short 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 in the program.

LOG (X)

This function calculates the natural logarithm (i.e., log to the base e) of any numeric expression X.

Example:

```
10  INPUT A
20  B = LOG (A)
30  PRINT B
40  END
RUN
? 3.2
1.1631508098056
***40 END***
```

MAX (X_1,...,X_n)

This function instructs the computer to examine a list of numeric expressions (X_1 through X_n) and return the value of the numeric expression with the maximum value.

Example:

```
10  X = 10
20  Y = 25
30  M = MAX (X,Y)
40  PRINT M
50  END
RUN
25
***50 END***
```

In this example, the maximum value contained within the list is 25.

MIN (X_1,...,X_n)

This function instructs the computer to examine a list of numeric expressions (X_1 through X_n) and return the value of the numeric expression with the minimum value.

Example:

```
10  X = 5
20  Y = 10
30  M = MIN (X,Y)
40  PRINT M
50  END
RUN
5
***50 END***
```

In this example, the minimum value contained within the list is 5.

RND (X)

This function generates a random number between 0 and 1. The numeric expression X is really a dummy expression which is required but which is not actually used in the generation of a random number.

Example:

```
10  FOR N = 1 TO 100
20  PRINT RND (2)
30  NEXT N
40  END
RUN
```

The above example will print out 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:

```
10  RANDOMIZE
20  FOR N=1 TO 100
30  PRINT RND (2)
40  NEXT N
50  END
RUN
```

SGN (X)

This function returns a +1 if the value of the numeric expression X is greater than Ø, a Ø if X equals Ø, and a -1 if X is less than Ø.

Example:

```
10  INPUT A, B, C
20  PRINT SGN (A)
30  PRINT SGN (B)
40  PRINT SGN (C)
50  END
RUN
?-12, 0, 14
-1
0
1
***50 END***
```

SQR (X)

This function calculates the square root of any positive numeric expression X.

Example:

```
10  INPUT A
20  INPUT B
30  LET M = SQR (A*B)
40  PRINT M
50  END
RUN
? 6
? 2
3.4641016151378
***50 END***
```

TRIGONOMETRIC FUNCTIONS

ATN (X)

This function calculates the arctangent, in radians or degrees of any numeric expression X.

Example:

```
10  PRINT ATN(.80)
20  RUN
RUN
.67474094222353
***20 END***
```

In this example the arctangent of .80 is equal to .67474094222353 radians.

If the DEG mode has been selected, the resulting value is expressed in degrees. Thus:

```
DEG
10  PRINT ATN(1.0)
20  END
RUN
45.000000000002
***20 END***
```

Although Cromemco 16K 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.)
```

COS (X)

This function calculates the cosine of an angle X, where X is any numeric expression. (See ATN(X) for a description of how to calculate ARCCOS(X).) Unless DEG mode has been selected, it is assumed that the value of X is expressed in radians.

Example:

```
10  INPUT A
20  LET B = A*2
30  PRINT COS (B)
40  END
? .60
0.36235775447529
***40 END***
```

In this example, the cosine of a 1.20 (=B) radian angle is .36235775447529.

Example:

```
DEG
10  PRINT COS (60)
20  END
RUN
0.49999999999949
***20 END***
```

The cosine of 60 degrees is 0.5. The result printed represents the computational accuracy of the 16K BASIC.

SIN (X)

This function calculates the sine of an angle X, where X is any numeric expression. (See ATN (X) for a description of how to calculate ARCSIN (X).) It is assumed that the value of X is expressed in radians unless the DEG mode has been specified.

Example:

```
10  INPUT A
20  PRINT SIN (A*3)
30  END
RUN
? .04
0.11971220728892
***30 END***
```

In this example, the sine of a .12 radian angle is approximately .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:

```
DEG
10  PRINT SIN (90)
20  END
1.0
***20 END***
```

The sine of a 90 degree angle is 1.0.

TAN (X)

This function calculates the tangent of any angle X, where X is a numeric expression. It is assumed that the value of X is expressed in radians unless the DEG mode has been specified.

Example:

```
DEG
10  PRINT TAN (30)
20  END
RUN
0.577350269189
***20 END***
```

The tangent of a 30 degree angle is .577350269189.

STRING FUNCTIONS

ASC (X$)

This function gives the equivalent ASCII decimal value of the first character of any string expression X$. An ASCII Character Set Table is included in Appendix B.

Example:

```
10  INPUT X$
20  PRINT ASC (X$)
30  END
RUN
? A
65
***30 END***
```

In this example, the ASCII decimal value for string character A is 65.

CHR$ (X)

This function gives a single character string which is the ASCII equivalent of the value of the numeric expression X, where $0 \leq X \leq 255$.

Example:

```
10  INPUT X
20  PRINT CHR$ (X)
30  END
RUN
?42
*
***30 END***
```

The ASCII decimal value 42 is equivalent to the character *. Thus, in the above example, the command CHR$ (42) instructs the computer to output the character * on a terminal.

This command allows the user to draw graphs or figures with special characters. The command may also be used to initiate special functions such as cursor positioning, generating line or form feeds, or causing a bell or some comparable device to sound on a terminal.

LEN (X$)

This command returns an integer value which is equal to the number of characters in any string variable X$. In other words, the LEN function gives the length of a string. Both characters and spaces are counted as part of the length.

Example:

```
10  DIM X$(20), Y$(30)
20  INPUT X$,Y$
30  PRINT "THERE ARE "; LEN(X$); " CHARACTERS IN STRING X$"
40  PRINT "THERE ARE "; LEN(X$); " CHARACTERS IN STRING Y$"
50  END
RUN
? EXAMPLE
?? LENGTH COMMAND
THERE ARE 7 CHARACTERS IN STRING X$
THERE ARE 14 CHARACTERS IN STRING Y$
***50 END***
```

POS (X$,Y$,n)

This command is used to locate the position within a string (X$) of the first character of a substring (Y$). The position within the string X$ at which the search is to begin is specified by the numeric expression n. This function gives a value equal to the position of the first character of the substring within the string.

Example:

```
10  DIM X$ (50)
20  X$ = "THIS IS A SUBSTRING SEARCH"
30  P = POS (X$,"IS",4)
40  R = POS (X$,"R",20)
50  PRINT P
60  PRINT 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 X$ and second to search for "R" starting from the twentieth character in X$. 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 P and 23 for R.

STR$ (n)

This function converts a numeric expression (n) into a string which is the character representation of the expression.

Example:

```
10  INPUT A
20  A$ = STR$ (A)
30  PRINT A$
40  END
RUN
? 8.45
8.45
***40 END***
```

In this example, the numeric value A = 8.45 is converted to the string "8.45".

VAL (X$)

This command converts a string variable (X$) into a number.

Example:

```
10  X$ = "26.6321"
20  A = 1
30  B = A + VAL (X$)
40  PRINT B
50  END
RUN
27.6321
***50 END ***
```

In this example, the string X$ is converted to its numeric equivalent (26.6321) so that this value can be added to the numeric expression A.

If the argument string for VAL consists of both numeric and non-numeric information:

a) if the first character is non-numeric, VAL will return a zero value (this can be an extremely useful way of decoding a user's input)

b) if the first character(s) is a number this will be converted without consideration of the portion of the string after the first non-numeric character.

PROGRAMMER DEFINED FUNCTIONS

DEF FNs (X_1, X_2,..., X_n) = Y

The DEF function permits the programmer to define up to 26 functions which are in addition to the pre-defined functions included in 16K BASIC. In the command, "s" is any single letter from A through Z, X_1 through X_n are any arithmetic "dummy" variables, and Y is any arithmetic expression.

User-defined functions are restricted to one line.

Example:

```
10  X = 20
20  DEF FNA (X) = (X*20 + 1.156)/3.14
30  PRINT FNA (X)
40  END
RUN
127.75668789809
***40 END***
```

2.6 ADVANCED PROGRAM INSTRUCTIONS, FUNCTIONS, AND EXAMPLES

Cromemco 16K Extended BASIC provides the advanced programmer with a number of instructions and functions which are designed to facilitate error handling capabilities during program execution, aid in debugging programs, allow machine language interface, and control output to a terminal. This section includes a description of each of these.

ECHO

This instruction is used to re-enable the display of information at a terminal after the display has been disabled by the NOECH instruction. ECHO may be used as either a command or as a statement. The general format for this instruction is simply:

ECHO

ESCape

ESC may be used as either a statement (with a line number) or as a command (without a line number). The ESC instruction is used to re-enable ESCape key operation after it has been disabled by the NOESC command. The general format for this instruction is simply:

ESC

INP, OUT

These instructions are used to INPut data from and OUTput data to a previously defined I/O port.

The INP function allows the user to read the contents of any I/O location. The general format for this function is:

```
INP (M)
```

where M is any I/O address.

The OUT instruction is used to output data to a given I/O location. The general format for this statement is:

```
OUT M,b
```

where M is any I/O address and b is the byte value to be output.

Example:

```
10  A = INP(%FF%)
20  PRINT A
30  OUT %FF%,A
40  END
```

MAT m = numeric expression

This statement is used to set all elements in a matrix (m) equal to the value of the numeric expression.

Example:

```
10  DIM A (1,4)
20  READ A(1,1), A(1,2), A(1,3), A(1,4)
30  DATA 20, 21, 22, 23
40  PRINT A(1,1), A(1,2), A(1,3), A(1,4)
50  MAT A = 0
60  PRINT A(1,1), A(1,2), A(1,3), A(1,4)
70  MAT A = 1
80  PRINT A(1,1), A(1,2), A(1,3), A(1,4)
90  END
RUN
20              21              22              23
0               0               0               0
1               1               1               1
***90 END***
```

NOECHO

The NOECHO instruction is used to disable the display of information at a terminal. This command is most commonly used when inputting information such as a password which the user needs to protect. The information display can be re-enabled on the terminal by using the ECHO instruction. The general format for the NOECHO instruction is simply:

NOECHO

NOESCape

NOESC may be used as either a command (no line number) or as a statement (with a line number). The purpose of the NOESC instruction is to disable the ESCape key operation on a terminal. Most terminal keyboards include a key labelled ESC which when pressed will abort program execution and return the terminal to text input mode. The NOESC instruction is used to prevent program interruption when the ESC key is pressed.

The general format for this instruction is:

NOESC

NTRACE

The NTRACE instruction disables the TRACE instruction which calls for line by line examination of a program during execution. NTRACE may be used as either a command (with no line number) or as a statement (with a line number). The general format for this command is simply:

NTRACE

ON ERROR

The purpose of this command is to direct the computer to a specific statement or routine when a non-fatal error occurs during program execution. The general form of this command is:

ON ERROR { STOP / GOTO / GOSUB } line number

where the line number is any line number in the current program. A non-fatal error in 16K BASIC is any error listed in the error table (see Appendix A) with a number of 128 or above. Errors numbered 127 and below are defined as fatal errors and cannot be trapped with an ON ERROR statement.

If ON ERROR is written at the beginning of a program, the statement specified with ON ERROR will be executed each time a program error occurs. If placed elsewhere in the program, the statement will be executed only for errors which occur during the execution of statements following the ON ERROR statement.

Example:

```
 60  INPUT A,B
 80  PRINT A*B
100  ON ERROR GOTO 300
120  INPUT C,D
140  PRINT C/D
160  GOTO 60
300  PRINT "NON-FATAL STRING ENTRY ERROR"
320  GOTO 120
```

In this example, any error which occurs before line 100 will be dealt with by the standard system error handling procedure. Any trappable error which occurs after line 100 will cause the program to execute statement line 300.

ON ESC

This instruction directs the program to execute a specified statement when the ESC key is pressed. The general form for this instruction is:

ON ESC { STOP / GOTO / GOSUB } line number

where the line number is any line number in the current program. Once the ON ESC instruction has been executed, program control can be passed to the specified statement when the ESC key is pressed. The statement specified in the ON ESC instruction is then executed.

Example:

```
10  INPUT A,B,C
20  LET D = A + 3.2
30  ON ESC GOTO 10
40  PRINT A
50  GOTO 50
60  END
RUN
```

In this example, if the user presses the ESC key after line 30 is executed, the program will be directed back to line 10 and the prompt will again appear requesting input data.

PEEK, POKE

Cromemco 16K BASIC provides a machine language interface through the PEEK function and POKE command. These two instructions are used to access the contents of memory locations directly from BASIC.

The PEEK function allows the user to read the contents of any memory address. The general format for this funciton is:

```
PEEK (M)
```

where M is any memory address.

If the statement A=PEEK(M) is coded, the contents of memory location M are examined and the value is assigned to the variable A.

The POKE instruction is used to insert a byte into a given memory location. The general format for this instruction is:

```
POKE M, b
```

where M is any memory address and b is any byte. The byte, represented by its hexadecimal equivalent, is POKEd into memory location M.

PEEK and POKE are used to store byte-oriented information. If M is too large to be treated as an integer, or if b is not in the range 0 to 255, an error message will result.

PRINT USING

The PRINT USING statement allows the user to specify a particular format for printing output. The general form of the PRINT USING statement is:

PRINT USING f, X_1, X_2,...,X_n

where f is a specified format which may include a number of special characters and string literals and X_1 through X_n are any numeric, string or subscripted variables or string literals.

All normal PRINT functions (such as TAB, SPC, semicolon, and comma) are overriden by the PRINT USING statement.

There are a number of special characters used for outputting numeric data in the PRINT USING statement. These special characters include:

#	&	*
	+	
	,	
	$	
	!	
	-	
	.	

A brief explanation of each of these special characters follows. The format expression may include a maximum of 128 characters. Note also that if a number being formatted has more digits than allowed for in the format expression, an all-asterisk error message will result.

Digit Formating

1) Digit Formating With Leading Blanks (#)

The # sign is 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 # 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 blanks.

Example:

In the statement PRINT USING "#####",A

if A=1 the output is bbbb1,

if A=12 the output is bbb12,

if A=123 the output is bb123

if A=123456 the error message *****is printed.

In this example, b represents a blank space (the b is not printed out).

2) Digit Formating with Leading Zeros (&)

The & character is also used to right justify digits in a print field. Any non-digits (such as a minus sign) are eliminated. If the number of & characters in the format field exceeds the number of digits in the expression, a Ø is printed in the extra spaces.

Example:

In the statement PRINT USING "&&&&&", A

if A=1 the output is ØØØØ1

if A=12 the output is ØØØ12

if A=123 the format is ØØ123

if A=123456 the error message ***** is printed.

3) Digit Formatting With Leading Asterisks (*)

The * is used to right justify digits in a print field. Any non-digits (such as a minus sign) are eliminated. If the number of characters in the format field exceeds the number of digits in the expression, an * is printed in the extra spaces.

Example:

In the statement PRINT USING "*****", A

if A=1 the output is ****1

if A=12 the output is ***12

if A=123 the output is **123

if A=123456 the output is *****.

4) Comma (,)

The "," character is used to place a comma in the position in which the character appears in a string of digits (#,&,or*) 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.

Example:

In the statement PRINT USING "##,###", A

if A = 2003 the output is b2,003;

if A = 4 the output is bbbbb4.

In the statement PRINT USING "&&&,&&", A

if A = 4457 the output is 044,57;

if A = 18 the output is 000,18.

In the statement PRINT USING "*****,*", A

if A = 996546 the output is 99654,6;

if A = 22 the output is ****2,2.

5) Decimal Point (.)

The character "." places a decimal point in the position in which the character appears in a string of digits (#, &, *) 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 available.

Examples:

In the statement PRINT USING "###.###", A

if A = 234 the output is 234.000;

if A = 23.4567 the output is b23.457.

In the statement PRINT USING "&&.&", A

if A = 13 the output is 13.0

if A = 66.72319 the output is 66.7

In the statement PRINT USING "*****.**", A

if A = 876.1245 the output is **876.12

if A = 1234567.245 an error message ******** results

Note in the last example that when too many significant digits to the left of a decimal point appear, an all-asterisk error message results.

6) 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 first 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:

In the statement PRINT USING "+##.###", A

if A = 56.8888 the output is +56.889

if A = 4.564 the output is +b4.564.

In the statement PRINT USING "+&&&.&&", A

if A = 6.456 the output is +ØØ6.46;

if A = 234.2 the output is +234.20

In the statement PRINT USING "-*****.*", A

if A = -23.56 the output is-***23.6

if A = 2345 the output is b*2345.0;

if A = -2345678.34 the error message ******** is printed.

7) Floating Plus (++) or Minus (--) Signs

The use of two or more plus or minus signs at the beginning of a format field will output the respective plus or minus sign directly preceding the value of the expression. If a positive expression is used with 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:

In the statement PRINT USING "++###.###", A

if A = 2.234 the output is b+b2.234

if A = 22.234 the output is b+22.234

In the statement PRINT USING "--&&&.&", A

if A = -44.56 the output is b-Ø44.6

if A = 5.32 the output is bbØØ5.3

In the statement PRINT USING "++****.**", A

if A = 178.456 the output is b+*178.46

if A = 12345678.45 an error message ********* is printed.

8) Dollar Sign ($)

The character $ 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:

In the statement PRINT USING "$##.##", A

if A = 23.456 the output is $23.46;

if A = 4.52 the output is $b4.52

In the statement PRINT USING "-$&&&.&&&", A

if A = 57.654 the output is b$Ø57.654;

if A = 123.7789 the output is b$123.779.

In the statement PRINT USING "$*.*", A

if A = 2.34 the output is $2.3;

if A = 234.55 an error message **** is printed.

9) 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 two or more dollar signs begin in the second character position, the first character position must contain a plus or a minus sign.

Examples:

In the statement PRINT USING "$$$$#.###", A

if A = 234.2345 the output is b$234.235

if A = 4.45 the output is bbb$4.45Ø

In the statement PRINT USING "$$&&.&&", A

if A = 23.989 the output is b$23.99

if A = 4.5 the output is b$Ø4.5Ø.

In the statement PRINT USING "-$$***.**", A

if A = 24.56 the output is bb$*24.56

if A = 4455.67 the output is b$4455.67

10) Exponent Fields (!!!!)

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

Example:

In the statement PRINT USING "##.##!!!!", A

if A = 23.3456 the output is 23.35E+ØØ

if A = 2000 the output is 20.00E+Ø2

In the statement PRINT USING "-&&&.&&!!!!", A

if A = -.36 the output is 3600.05E-Ø3

A format field is bounded on either side by any character which is not one of the special characters defined in the PRINT USING statement. As indicated in the general format for PRINT USING statements, 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 be assigned to a string variable.

When a string literal appears in a format expression, the characters of the string literal 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 literal is less than the number of characters in the format field, the extra spaces will be left blank. If the number of characters in the string literal is greater than the number of characters in the format field, the extra characters in the string literal will be truncated.
Example:

In the statement, PRINT USING "****,**.*", "STRING"

the string literal is output in the format STRING

In the statement, PRINT USING "&&&,&&&.&&", "STRING-LITERAL"

the string literal is output in the format STRING-LIT

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.

Example:

In the statement, PRINT USING "$$&&.&&", A,B,C
the expressions A, B and C will all be formatted with the format field $$&&.&&

In the statement, PRINT USING "$$##bb$$&&.&", A,B,C
the expressions A and C will be formatted using the format field $$## and the expression B will be formatted using the format field $$&&.&

There are a number of general rules in 16K BASIC which the user should keep in mind when formatting with the PRINT USING statement. These rules are as follows:

-- Only the characters # or & should be used to the right of a decimal point. The asterisk (*) should not be used to the right of a decimal point.

-- Either the floating dollar ($$) character or pluses (++) or minuses (--) may be used within the same formatting statement, but not both types of characters.

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

-- The asterisk (*) character follows the same rules as the # character when used to the left of a decimal point.
-- 16K BASIC does not check comma syntax.
-- Only one decimal point may be used in a format field.
-- Trailing + or - signs are illegal in the exponential format (i.e., E+nn.)

A typical application for the PRINT USING format follows. This application is a program for printing payroll checks on an automated basis.

```
 10  PRINT "ENTER SOCIAL SECURITY NUMBER AND HOURS
     WORKED"
 20  PRINT "OF FOUR EMPLOYEES"
 30  PRINT
 40  INPUT A, A1
 50  INPUT B, B1
 60  INPUT C, C1
 70  INPUT D, D1
 90  PRINT "WHAT IS THE HOURLY WAGE?"
100  INPUT W
110  PRINT
120  PRINT
130  A1 = A1*W : B1 = B1*W : C1 = C1*W : D1 = D1*W
```

```
140  PRINT A, B, C, D
150  PRINT
160  PRINT USING "$$$#.##bbbbbbb", A1, B1, C1, D1
170  PRINT
180  PRINT "ANOTHER FOUR? TYPE 1 IF YES, TYPE Ø IF NO"
190  INPUT Q
200  IF Q = 1 THAN GOTO 10
210  END
RUN
ENTER SOCIAL SECURITY NUMBER AND HOURS WORKED OF FOUR EMPLOYEES
?552966937
??40
?525449121
??40
?455238541
??35
?211329494
??32
WHAT IS THE HOURLY WAGE?
?4.00
552966937        525449121       455238541       211329494
  $160.00          $160.00         $140.00         $128.00
ANOTHER FOUR? TYPE 1 IF YES, TYPE Ø IF NO
?0
END
```

SET

To change the default values of system functions, the user can specify the following instruction:

SET n,v

where n is the number of the system function to be changed (see SYS (X)) and v is the new value for the function. Example:

SET Ø,130

changes the page width to 130 characters.

A special use of the set command is

SET 0,-1

which inhibits automatic carriage return-line feed at the end of a line.

Using SET Ø,-1 to disable page width checking is especially useful for graphics output to Qume-type printers.

For disk BASIC only, a SYS entry has been added to facilitate timed input. To start the timer, the programmer codes

SET 5,V

where V is approximately ten (1Ø) times the number of seconds he wishes to delay. (Double V if using a 2MHz clock! Experiment if using slow memory boards.)

When the next INPUT statement is encountered, BASIC will issue an "ERROR 210 -- INPUT TIMEOUT" if the user does not respond with a complete input within the allotted time!

The programmer may find how much time was used by coding SYS(5) to find time remaining. The timeout is active at all following INPUT statements, so when it is no longer desired it may be de-activated by coding

SET 5,Ø .

The programmer may use the ON ERROR statement to trap the timeout error.

SPC (X)

The SPC function is used only in a PRINT statement and is used to instruct the computer to leave a specified number of spaces. The general format for this function is:

SPC (n)

where n is any integer value. Unlike the TAB function, which always determines print position relative to column Ø, the SPC command determines print position relative to the current column location.

Example:

```
10  A = 2
20  INPUT C,D
30  PRINT SPC(10);A;SPC(A*10);C;D
40  END
RUN
?20,30

          2                    2030
          ↑                    ↑
column    10                   31
  no.
```

In this example, the computer is instructed to skip 10 spaces, print A, skip 20 spaces, and print C and D.

SYS (X)

This function provides system information based on the value of the numeric expression X.

Example:

```
SYS (0) = page width
SYS (1) = tab field width
SYS (2) = last printed (i.e., current) character
SYS (3) = last runtime error number (for disk BASIC)
        = last runtime error two-letter ASCII code
          (for stand-alone BASIC)
SYS (4) = current print column number
SYS (5) = current input timer value
```

See the SET instruction for methods of changing the default values.

TAB

The TAB function is used only in a PRINT statement and is used to instruct the computer to begin printing at a specified column number. The general format for this command is:

```
TAB (X)
```

where X is any numeric expression.

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

Example:

```
10  A = 1
20  INPUT B,C,D,E
30  PRINT TAB (2);B;TAB(5);B+C+D;TAB(A+9);D
40  END
RUN
?5,8,9,4
  5  22   9
```

In this example, the computer is instructed to begin printing B in column 2, to begin printing B+C+D in column 5, and to begin printing D in column 10. Note that columns are numbered 0 through the page width so the first column is column 0.

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 page-width. The default value for the page width is 79.

If the argument is negative, no tabbing takes place (i.e. same as SPC(Ø)).

If the (reduced) argument is greater than the current column position, a new line (CR,LF) is issued and the TAB is executed on the next line.

Note: If an SPC or TAB function is used in places other than a PRINT statement, the functions are "no-ops" -- that is, they do not alter their arguments in any way.

Example:

A = SPC(9.5) is the same as A = 9.5

TRACE

The TRACE instruction allows the user to follow the execution of a program line by line by directing the computer to list the line number of each statement being executed. Statement line numbers are listed in brackets.

Example:

```
10  TRACE
20  INPUT X
30  PRINT "THIS IS"; X
40  LET Y = X + 1
50  PRINT Y
60  END
RUN
<20>?  10
<30> THIS IS 10
<40>
<50>
***60 END***
```

USR

The USR function makes it possible to call an assembly language subroutine from a 16K BASIC program. The general form of the USR function is:

USR (address, p_1, ... , p_n)

where the address is the address of the assembly language subroutine and parameters p_1 through p_n are converted to 16-bit integers.

The USR function always requires the user to specify one parameter, even if it is a dummy parameter. For example, USR(Ø,1) is correct, while USR(Ø) will result in a syntax error.

When the user routine gains control (at the address specified in the USR function call), the following conventions apply:

1) Register A contains the number (n) of parameters in the function call.

2) Register pair HL contains the "return" address to BASIC. The user routine may re-enter BASIC via JP (HL).

3) The parameters are placed in order on the CPU stack and may be recovered via POP instructions.

4) <u>If and only if</u> n parameters (n is the contents of register A, as above) are POPped off the stack, BASIC may be re-entered via a RET instruction.

5) The routine may return a 16-bit value to be assigned to the function by placing the value in register pair DE before re-entering BASIC.

6) Aside from the restrictions noted above, all registers may be used in any way by the user routine.

3.0 FILE ORGANIZATION

File Definition and Use

A file is a string of bytes which is usually stored on a medium such as a disk or paper tape and which is given a file name. The file name is used to refer to or call the file. A file may consist of any string of bytes including, for example, a program, a list of data, or any body of text. BASIC files may be read from or written to a number of devices including the disk, punch, paper tape, printer, etc.

Applicability

All of Chapter 3 of this manual is applicable to 16K BASIC run under CDOS. **Only the following** instructions in Chapter 3 can be used with Stand Alone BASIC:

OPEN	PRINT
CLOSE	INPUT
PUT	LIST
GET	ENTER
IOSTAT	BYE

File Naming Conventions

Disk Files

In Cromemco 16K BASIC, disk file names specify both the disk drive and the file name plus an optional extension.

The general format for disk file names is as follows (square brackets refer to optional quantities):

[X:] file name [.extension]

where X: is an optional disk drive specifier (i.e., A:, B:, C:, D:), the file name is a 1 to 8 character file name, and .ext is an optional 1 to 3 character extension to a file name. Both the file name and the extension may include any printable ASCII character with the exception of the following:

$ * ? = . , : space

Non-Disk Files

The general format for non-disk files and device drives is a three character name (XXX) where the first character in the name is equal to a $. The last two characters specify a particular device driver. Neither of the last two characters may be a colon (:). More than two characters may follow the $ character if the user feels the extra character will assist in defining a device driver more clearly. However, only the first two characters are significant. If the first character is not a $, a CDOS disk file is assumed.

I/O Channels

16K BASIC is supplied with 4 I/O channels in addition to the console (each I/O channel occupies 192 bytes of memory.) To change the number of I/O channels see Appendix C.

Device Drivers

There are two device drivers supplied with both Disk and Stand Alone BASIC: $SY (for console I/O) and $T5 (see Appendix

C: TU-ART I/O Port Driver). In addition, Disk BASIC supplies the following drivers: $PU (punch), $RD (paper tape reader), and $LP (line printer).

Random Access Files

Cromemco 16K BASIC provides the user with both sequential and random access. Random access describes the process of obtaining information from a disk or other storage device where the time required for such access is essentially independent of the location of the information most recently obtained or placed in storage.

The following program demonstrates the random access capability of Cromemco BASIC:

```
 100  INPUT "RECORD SIZE ?? ",R
 200  DIM A$(R-1)
 300  DIM B$(40)
 400  INPUT "FILE NAME ?? ",B$
 500  OPEN\1,R\B$
 600  INPUT "RD=1,WR=2 >> ",Q
 700  INPUT "RECORD # , BYTE # >>> ",R,B
 800  IF Q=1 THEN 1100
 900  IF Q=2 THEN 1500
 950  CLOSE\1\
1000  STOP
1100  REM RD
1200  GET\1,R,B\A$(-1)
1300  PRINT A$
1400  GOTO 600
1500  REM WR
1600  INPUT "DATA ..>> ",A$
1700  PUT\1,R\A$(-1)
1800  GOTO 600
1900  END
```

In the example listed above, the user should note the following important points:

1) In line 100, note that BASIC allows any arbitrary record size from 1 to 32767 bytes.

2) In line 200, note that DIMs may use expressions

as arguments.

3) In line 500, note the power of being able to use any string as a file name. File B$ is opened for reading and writing with a record length of R.

4) In line 950, note that if you do not CLOSE a file you may lose some or all of the data written to that file.

5) In line 1200, note that starting at byte number B within record number R, we read a string of bytes which fill A$ (recall A$(-1) implies A$(Ø, DIM of A$)).

6) In line 1600, note that if you do not fill A$ with Input at that point, BASIC fills the rest of A$ with nuls.

7) In line 1700, note that starting at byte zero (the byte number defaults to Ø if omitted) of record number R, we write the "full" A$ to the file thus filling the record.

The user should keep in mind that a file must be CREATEd before the file can be OPENed.

3.1 FILE STATEMENTS

Brief descriptions of each of the file statements included in Cromemco 16K BASIC are presented in the following section.

OPEN

The OPEN statement allows the user to link a file or a system device with a file number for future reference in connection with file I/O statements.

The general form for the OPEN statement is:

OPEN\file number, p_1, p_2\string expression

The user should note that the general form for these file statements includes a file number and two parameters (p_1 and p_2). The file number ranges from 1 to the maximum channel available. The maximum channel available is installation dependent. The use of a file number out of range will result in an error message.

Specification of p_1 and p_2 is optional and these parameters may be used either singly or in combination. For disk files only, the following definition holds:

On OPEN, p_1 specifies record size. Record size may be any value from 1 to 32767. If not specified, the default value assigned to parameter 1 is 128 bytes per record. Parameter 2 (p_2) specifies read/write access to a file. Values assigned to p_2 may be 1, 2, or 3, where 1 equals read, 2 equals write, and 3 equals both read and write. The default value for p_2 is 3 (read and write).

The string expression may be any file name.

Example:

```
10  OPEN\1\"$LP"
```

This statement instructs the computer to open file 1 for access to the line printer.

[Note that although files OPENed for READ/WRITE may be used as WRITE-ONLY files, slightly faster execution speeds may occur if the file is OPENed as WRITE-ONLY.]

CLOSE

This command allows the user to disassociate a file and a file number so that the file cannot be referenced with that number. The general form of this statement is:

CLOSE[\file number\]

The file number is optional but if used may range from 1 to the maximum channel available. The number Ø cannot be used as a file number.

On CLOSE, parameters 1 and 2 have no meaning.

If the file number is <u>not</u> specified, <u>all</u> active files are closed.

Example:

```
10  CLOSE\1\
```

This statement instructs the computer to close file 1.

```
900  CLOSE
```

The statement in line 900 instructs the computer to CLOSE all files currently OPEN.

PUT

The PUT command allows the user to write data in binary format (see Notes on Binary Format at the end of this section) into a file. The general form of this command is:

PUT\file number, p_1, p_2\[exp_1,...,exp_n]

where exp_1 through exp_n may be one or more numeric expressions, numeric or string variables, or string literals. If no expression follows the second backslash, only device status is set (e.g., record and byte position on a disk file). This is a useful way of setting status (position) without actually initiating any data transfer.

For disk files only, p_1 is equal to the record number and p_2 specifies the byte number. If parameter 1 is not specified, then the default procedure is sequential access. If parameter 1 is specified and parameter 2 is omitted, then p_2 defaults to Ø. Specifying a negative number for either parameter will result in the default value being assigned to that parameter.

For non-disk files, p_1 and p_2 may be optionally required and/or used by some device drivers.

GET

The purpose of the GET command is to get (or read) data in binary format (see Notes on Binary Format at the end of this section) from a file. The general form for this command is:

GET\file number, p_1, p_2\variable expression list

The variable expression list may consist of one or more numeric or string variables. These variables are assigned values which are read sequentially from a file so each variable type (string or numeric) must correspond to the data type being read. For disk files only, p_1 is equal to the record number and p_2 specifies the byte number. If parameter 1 is not specified, then the default procedure is sequential access. If parameter 1 is specified and parameter 2 is omitted, the p_2 defaults to Ø. Specifying a negative number for either parameter will result in the default value being assigned to that parameter. For non-disk files, p_1 and p_2 may be optionally required and/or used by some device drivers.

Example:

```
DIM A$ (31), B$ (20)
GET\F,R,B\A$(-1), B$(10,19)
```

In this example, the GET statement requests that, starting at byte B of record R in file number F, 32 bytes be read into string A$ and then 10 bytes are read into B$ starting at byte number 10 of B$. (A$(-1) specifies A$(Ø, DIM of A$ => A$(Ø,31).)

Notes on "Binary Format"

PUT and GET write and read files where the content of the file is assumed to be consistent with BASIC's internal numeric and string representations. Thus, numeric items occupy space in the file according to the following table:

Integer Items -- 2 bytes each

e.g., PUT\4\7,9

would write 4 bytes

Short Floating Point Items -- 4 bytes each

e.g., SHORT A, B(10)

GET\3\B(1),B(2),A

would read 12 bytes

Long Floating Point Items -- 8 bytes each

e.g., PUT\2\SIN(30), SQR(A*A+B*B)

would write 16 bytes

String items occupy only as many characters in the file as would be moved if they were used on the right side of an equal sign in a LET statement.

Examples:

1) PUT\4,R,B\CHR$(255) would write 1 byte

2) GET\2\A$(Ø,11) would read 12 bytes

3) DIM B$(31)
PUT\4,9*Q+1\B$(-1) would write 32 bytes (see rules on string subscripts)

4) C$ = STR$(7.Ø/2)
PUT\1\C$ would write 3 bytes ("3.5")

PRINT

The PRINT statement is used to output data to an ASCII device (such as a line printer) or disk file by writing in ASCII into a sequential or random file.

The general form for this statement is:

PRINT\file number, p_1, p_2\ [USING] exp_1,...,exp_n

Exp_1 through exp_n may be a list of one or more numeric or string expressions, string or numeric variables, or string literals. Expressions in the list must be separated by either semicolons or commas. The output format is identical to the format instructions described under the general PRINT and PRINT USING statements. For disk files only, p_1 is equal to the record number and p_2 specifies the byte number. If parameter 1 is not specified, then the default procedure is sequential access. If parameter 1 is specified and parameter 2 is omitted, the p_2 defaults to Ø. Specifying a negative number for either parameter will result in the default value being assigned to that parameter. For non-disk files, p_1 and p_2 may be optionally required and/or used by some device drivers.

INPUT

The purpose of the INPUT command is to read data in ASCII from a file. The general form of the command is:

INPUT[\file number, p_1, p_2\][string expression]
variable expression$_1$,...,variable expression$_n$[;]

The user may include either the file number and parameters or the string expression (or neither) but not both (square brackets refer to optional quantities). Variable expressions may be one or more numeric and/or string variables whose values are read from a file or the terminal. The variable type must correspond to the data type being INPUT. Expressions in the variable list must be separated by commas. For disk files only, p_1 is equal to the record number and p_2 specifies the byte number. If parameter 1 is not specified, then the default procedure is sequential access. If parameter 1 is specified and parameter 2 is omitted, the p_2 defaults to Ø. Specifying a negative number for either parameter will result in the default value being assigned to that parameter. For non-disk files, p_1 and p_2 may be optionally required and/or used by some device drivers.

Examples:

1) INPUT\1\A,B,C$

In this example, the INPUT command instructs the computer to read 7-bit ASCII data from file 1 into the variables A, B, and C$.

2) INPUT A

The absence of the file number parameter indicates that input is to be read from the terminal. BASIC prompts the user with a question mark and the entered numeric value is placed in variable A.

3) INPUT "FILE NAME? >>", F$

BASIC prompts with the string in quotes instead of the question mark. The user's ASCII response is placed in variable F$.

4) INPUT "PASSWORD >>", P$(4,7);

BASIC will print the prompt PASSWORD >> and then place the first 4 characters of the user's response in positions 4 through 7 of variable P$. The trailing semicolon suppresses the echo of the user's input carriage return.

Using the "INPUT" statement will trap the following control characters and will not transmit them any further:

Control - J (Line feed)

Control - L (Form feed)

Control - M (Carriage return)

Control - U (Delete line)

Control - Z (End of file)

Control - [(Escape)

To circumvent this when using keyboard entry, the following statements may be used and will transmit all characters:

```
10  NOESC
20  OPEN\1\"$SY"
30  GET\1\A$(0,0)   :REM GET ONE CHARACTER
40  PRINT A$(0,0)   :REM PRINT IT (FOR EXAMPLE)
```

3.2 FILE INSTRUCTIONS

BYE

The BYE command is used to get the user out of BASIC and back into CDOS on disk systems or MONITOR on non-disk systems. The general form of this command is simply:

```
>>BYE
```

For disk systems, after the BYE command is typed in, the computer will respond with the current disk drive.
Example:

```
>>BYE
B.
```

In the example above, the prompt which appears after BYE is typed in and the user hits carriage return is the CDOS prompt of the current disk drive "B.".

For non-disk systems, BYE goes to location EØØ8H in the MONITOR.

CREATE

The purpose of this command is to CREATE a CDOS disk file (see the Cromemco Disk Operating System User's Manual). The general form of this command is:

CREATE <string expression>

where the string expression is any valid file name. Example:

CREATE "B:PAYROLL.NEW" will create the file PAYROLL.NEW on the disk in drive B.

NOTE: No space is allocated to a file by the CREATE command since all file space under CDOS is dynamically allocated only when and where needed.

An error message results if the file already exists.

DIR

The DIR command corresponds to the CDOS DIR command (see the CDOS User's Manual for a full description). The DIR command lists disk files giving size (in K-bytes) and number of extents. However, in BASIC, the file specifier (if used) must be enclosed in quotation marks or must be another valid form of string expression.

Examples:

```
DIR -- will list all files on the current disk.
DIR "A:*.*" -- will list all files on drive A.
DIR "*.SAV" -- will list all files on the current
               disk with the extension "SAV".
```

DSK

The DSK instruction prints the current disk drive. The general form of the instruction is:

DSK or DSK "d:"

where d: is the drive (i.e., A,B,C,D). DSK "d:" changes the default disk drive to the specified one. The d may also be a byte value of 1, 2, 3, or 4 and the colon may be omitted. For instance, DSK CHR$ (2) is equivalent to DSK "B:". Note also that IOSTAT (Ø,Ø) will return the number of the current disk (1=A:, 2=B:, 3=C:, 4=D:).

DSK "@" will log in a new disk after it has been inserted into the disk drive (but will remain in BASIC) always making Disk A the current drive. This is equivalent to typing Control-C while in CDOS.

ENTER

The ENTER command is used to enter statement lines from an external device or a disk file into the current program storage area. The user should keep in mind that statement lines being entered will replace current program statement lines if the entering line numbers are the same as the current line numbers. This command is frequently used when a programmer wishes to merge two programs input at different times into a single program.

The general format for this command is:

ENTER <string expression>

where the string expression is a file name.

Example:

ENTER "$RD" would enter the program listing found on the reader device.

When using ENTER to overlay portions of running programs, like line numbers in various overlays will cause most efficient memory usage.

Caution: The ENTER statement can only be terminated by certain errors and/or by receiving an ESCape request from the console or the program being read in.

ERASE

The purpose of the ERASE command is to remove a file from the directory. The general form of this command is:

ERASE <string expression>

where the string expression is any valid file name. All files whose names match the specified file name will be deleted.

Examples:

ERASE "DEMO" -- In this example, the file on the current disk with the name DEMO will be deleted.

ERASE "A:*.SAV" -- In this example, all files on disk A with extensions "SAV" will be deleted.

IOSTAT

The purpose of the IOSTAT function is to find the current status of a file number specifier (where there can be several different status values returned). The general form of this function is:

IOSTAT ($aexp_1$,$aexp_2$)

where $aexp_1$ is the file number and $aexp_2$ is a specifier. The specifier requests the desired status value.

In particular, for disk files only:

1) IOSTAT (F,Ø) will return a 1 if an end of file is detected on file F and a 2 if attempting to read an unwritten file segment in random access. A Ø is returned as normal status.
2) IOSTAT (F,1) will return the current sector number of file F.
3) IOSTAT (F,2) will return the current byte number within sector number IOSTAT (F,1).

Other device drivers may or may not return a status value.

LIST

The LIST command instructs the computer to print out one or more statement lines to a file or file device. The command may be used to output an entire program, a block of statement lines within a program, or a single statement line.

The general format for this command is:

LIST [string expression,] [n_1[,n_2]]

where the string expression is the name of a file and n_1 and n_2 are statement numbers in the file. When n_1 and n_2 are not specified, the LIST command will output the entire program. If both n_1 and n_2 are specified, the computer will list line number n_1 through line number n_2. If only n_1 is specified, the program is listed from line n_1 to the end of the program.

When the LIST command is used, the current program is automatically output to a file or file device in ASCII format. A program output by the LIST command can be read back into storage by the ENTER command.

Example:

```
LIST "LISTDEMO", 20, 200
```

In the example above, the computer is instructed to list lines 20 through 200 to the file named "LISTDEMO".

LIST outputs an ASCII ESCape character as the last output byte to enable files to be re-ENTERed properly.

[Note: SAVEd programs may or may not be compatible between various revisions of 16K BASIC. LISTed programs always are!]

LOAD

Once a program has been SAVEd in binary format, the LOAD command can be used to place the program into core storage. In other words, the LOAD command reads a program, which has already been SAVEd on an external storage device (or disk file) into memory.

The general format for the LOAD command is simply:

LOAD <string expression>

where the string expression is the name of any valid device or disk file.

Note: The LOAD instruction clears the user area before moving the called program into the user area. It cannot be used to merge or concatenate files. The ENTER instruction must be used for these purposes.

RENAME

The purpose of the RENAME command is to give a new file name to a file already in the directory. The general form of the RENAME command is:

RENAME filename$_1$, filename$_2$

where filename$_1$ is the existing file and filename$_2$ is the new name.

Example:

RENAME "DEMO","DEMOREN" -- In this example, the file name DEMO will be replaced by the file name DEMOREN.

RUN

The RUN command instructs the computer to execute a specified program. The general form of this command is:

RUN<string expression>

where the string expression is the name of a file. If a file name is not specified, the computer will execute the program currently in memory.

Examples:

RUN -- starts execution of the program currently in memory.

RUN "B:WUMPUS.SAV" -- LOADs program WUMPUS.SAV from disk B into memory and then begins execution.

BASIC may be entered from CDOS and requested to immediately RUN any SAVEd program by entering:

BASIC <program name>

at the CDOS command processor's prompt.

To EXECUTE a BATCH command from BASIC, PRINT the desired commands to a file called "A:BASIC.CMD" and then execute the following program:

```
100  CREATE "A:$$$$.CMD"
200  OPEN\1\"A:$$$$.CMD"
300  PUT\1\CHR$(11),"@ BASIC.CMD$"
400  CLOSE
500  DSK "A:"
600  BYE
```

You must specify the appropriate disk drive because "$$$$.CMD" appears as a peripheral device to BASIC. "B:" could have been used throughout the program above instead.

SAVE

The SAVE command instructs the computer to store the current program on disk or some other storage medium. The computer "saves" a program by writing the program in internal binary format to the disk or storage device.

The general form for the SAVE command is simply:

SAVE <string expression>

where the string expression specifies any valid device or disk file name.

Examples:

SAVE "PGM2.SAV" -- instructs BASIC to save the program in a disk file named PGM2.SAV on the current disk.

SAVE "$PU" -- instructs the BASIC to save the program on a punch device.

[Note: SAVEd programs may or may not be compatible between various revisions of 16K BASIC. LISTed programs always are!]

4.0 EXAMPLES OF 16K BASIC PROGRAM STRUCTURE

This section includes several examples of programs written in 16K BASIC. Working through one or more of the examples should provide the beginning programmer with a better understanding both of how a problem is translated into a computer program and of programming logic.

4.1 ACTIVE BANDPASS FILTER CALCULATION

One of the most elementary applications for a computer is to use the computer to do calculations that can be done on sophisticated hand calculators. A program written to perform such calculations can be saved and reused as often as required. An example of a simple calculation program follows. This program allows the user to input data on certain parameters associated with the design of an active bandpass filter based on a common operational amplifier. The program then calculates the necessary component values based on the data.

The first three statement lines of the program are as follows:

```
10  PRINT
20  PRINT "THIS PROGRAM CALCULATES DATA NEEDED TO
    CONSTRUCT";
30  PRINT "ACTIVE BANDPASS FILTERS USING 741-TYPE
    OP-AMPS."
```

The principal function of PRINT statements 20 and 30 is to document the program so that it can be run in the future by users who are unfamiliar with the program. All programmers should be encouraged to document programs thoroughly.

The next step in the program is to request the user to input a number of values for various parameters. Note that each variable is defined in preceding print statements.

```
 40  PRINT
 50  PRINT "WHAT IS THE CENTER FREQUENCY OF THE PASS
     BAND, IN";
 60  PRINT "HERTZ (e.g., 4000, 250, 60)"
 70  PRINT
 80  INPUT F
 90  PRINT
100  PRINT "WHAT IS THE DESIRED GAIN, IN DECIBELS?";
110  PRINT "(e.g., 0, 5, 25)"
120  PRINT
130  INPUT H
140  PRINT
150  PRINT "WHAT IS THE DESIRED Q OF THE FILTER?"
160  PRINT
170  INPUT Q
180  PRINT
190  PRINT "SELECT A CONVENIENT STARTING VALUE C
     FOR CAPACITORS C1 AND";
```

```
200  PRINT "C2.   IF  THE  VALUE  IS  IN  PICOFARADS,
     ENTER THE DATA";
210  PRINT "IN THE FORMAT: X...E-12.  IF THE VALUE
     IS MICROFARADS,";
220  PRINT "USE THE FORMAT:  X...E-6."
230  PRINT
240  INPUT C
```

In this schematic formula, C1 and C2 are equal. Once a value for C is specified, the resistor values can easily be found. The next step then is to specify the formulas for each resistor value, as indicated below:

```
250  PRINT
260  PRINT
270  PRINT
280  PRINT  "RESISTANCE IN OHMS"
290  PRINT
300  PRINT
310  LET W = F*2*3.14159
320  PRINT "R1", Q/(H*W*C)
330  PRINT "R2", Q/(((2*Q*Q)-H)*(W*C))
340  PRINT "R3", (2*Q) / (W*C)
350  PRINT "R4", 2* ((2*Q) / (W*C))
```

These formulae for R1, R2, R3, and R4, specify the actual calculations which must be performed after the parameter values are input in order to determine the various resistances. The final section of the program provides a simple routine which allows the user to easily re-enter the program and input a new set of values.

```
360  PRINT
370  PRINT
380  PRINT "WOULD YOU LIKE TO CALCULATE ANOTHER FILTER?"
390  PRINT "TYPE Y FOR YES."
400  PRINT
410  INPUT A$
420  PRINT
430  PRINT
440  PRINT
450  IF A$(Ø,Ø)="Y"  THEN GOTO 40
460  END
```

If the user does not wish to do another filter calculation and consequently types in a string starting with a character other than Y for A$, then the program ends and the terminal returns to the text input mode for 16K BASIC.

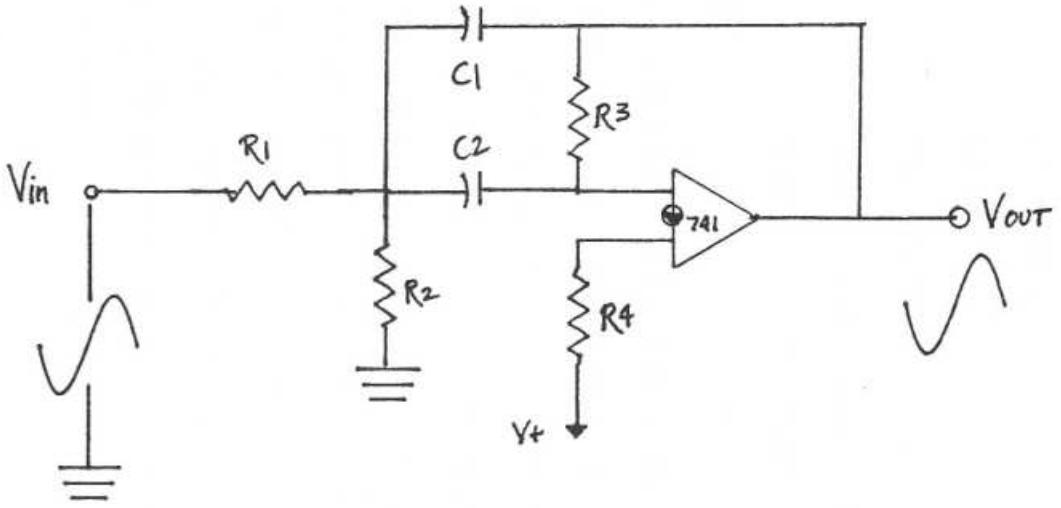
Note that the variables in this program are given letter symbols that correspond to their function. For example, it is surely easier to remember that the 'Q' of the filter is represented by variable Q than by variable L. Similarly,

the variable for capacitors C1 and C2 is C, rather than X or Y. These may seem to be intuitively obvious at this level of complexity, but as the program complexity increases the assignment of appropriate variable names becomes much more difficult.

Once the program has been input, we can instruct the computer to execute the program by typing in RUN. The following output will be generated:

```
RUN


THIS PROGRAM CALCULATES DATA NEEDED TO CONSTRUCT
ACTIVE BANDPASS FILTERS USING 741-TYPE OP-AMPS.

WHAT IS THE CENTER FREQUENCY OF THE PASS BAND, IN
HERTZ (E.G., 4000, 250, 60)

?1000

WHAT IS THE DESIRED GAIN, IN DECIBELS?

?20

WHAT IS THE DESIRED Q OF THE FILTER?

?100

SELECT A CONVENIENT STARTING VALUE C FOR CAPACITORS C1 AND
C2.  IF THE VALUE IS IN PICOFARADS, ENTER THE DATA
IN THE FORMAT:  X...E-12.  IF THE VALUE IS MICROFARADS
USE THE FORMAT:  X...E-6.

?1.0E-6

RESISTANCE IN OHMS

R1              795.77538762219
R2              5.0000314160973E-03
R3              31831.015504887
R4              63662.031009774
```

```
WOULD YOU LIKE TO CALCULATE ANOTHER FILTER?
TYPE Y FOR YES.

?NO

***460 END***
```



A BANDPASS 741-TYPE ACTIVE FILTER
USED IN PART 3.1 16K BASIC PROGRAMMING
AND APPLICATIONS

7-7-77 MA

4.2 STATISTICAL ANALYSIS PROGRAM

Another common use for a computer is the calculation of statistics for a given set of data -- particularly very large data sets. The program listed below calculates six common statistics for a given data set.

One of the first steps that must be taken in any program is the assignment of variable names to any variables which will appear in the program. The following program involves six functions which will require variable names. These functions and the corresponding variable name used in the program to represent each function are listed below:

Variable Name	Function
N	Number of elements in a data set
S	Sum of the numbers in a data set
T	Sum Squares
M	Mean
V	Variance
D	Standard Deviation

We can instruct the computer to calculate the number of elements in a data set by including a "counter" variable in the program. In this case, the "counter" variable is N. To count the number of elements, we set N equal to Ø initially and then increment N by 1 each time a new element in the data set is READ. Consequently, the first statement lines are used to initialize (set equal to Ø) certain variables.

```
 10  LET N = Ø
 20  LET S = Ø
 30  LET T = Ø
```

Note that S and T are also set equal to Ø. This will allow us to sum data elements and to sum the squares of data elements as these elements are READ. We can now begin reading in data elements.

```
 40  READ X
 50  IF X = 999 THEN GOTO 100
 60  LET N = N + 1
 70  LET S = S + X
 80  LET T = T+(X*X)
 90  GOTO 40
100  IF N = Ø THEN GOTO 300
```

The value 999 is used as a "dummy" value to indicate the end of a data set. The program continues to read in data and increment the values of the variables N, S, and T until a 999 is read from a DATA statement. Once a 999 value is read in, program control is transferred to statement line 100.

Statement line 110 through 130 compute the remaining three statistics.

```
110  LET M = S/N
120  LET V = (N*T-S*S)/N/(N-1)
130  LET D = SQR (V)
```

Next we include a set of titles to be printed out with each statistic.

```
140  PRINT
150  PRINT
160  PRINT "NUMBER", "SUM", "SUM SQUARES"
170  PRINT N,S,T
180  PRINT
190  PRINT "MEAN", "VARIANCE", "STANDARD DEVIATION"
200  PRINT M,V,D
210  PRINT
220  PRINT
230  PRINT
240  PRINT
```

At this point, all the statistics have been calculated for a set of data. Program control is therefore returned to the first statement in the program for that the next set of data can be read.

```
250  GOTO 10
260  DATA 1,2,3,4,5,6,7,8,9,10,999
270  DATA 2.3,2.4,2.5,2.6,2.7,2.8,999
280  DATA 10,15,29,28,12,44,5.5,2.2,999
290  DATA 999
300  END
```

Once this program has been entered, edited, and examined for errors, it can be RUN. The results are as follows:

```
RUN

NUMBER          SUM             SUM SQUARES
10              55              385
MEAN            VARIANCE        STANDARD DEVIATION
5.5             9.1666666666666 3.0276503540975

NUMBER          SUM             SUM SQUARES
6               15.3            39.19
MEAN            VARIANCE        STANDARD DEVIATION
2.55            0.035           0.18708286933869
NUMBER          SUM             SUM SQUARES
8               145.7           4065.09
MEAN            VARIANCE        STANDARD DEVIATION
18.2125         201.64696428571 14.200245219211

***300 END***
```

4.3 DEMONSTRATION PROGRAM

The program listed on the following page demonstrates BASIC's abilities to PEEK, accept HEX input, and perform logical operations. This program also uses a wide range of BASIC commands which the user is encouraged to examine carefully.

```
10 REM
20 REM DEMONSTRATION PROGRAM ---- PRODUCES HEX MEMORY DUMP
30 REM
40 PRINT"THIS PROGRAM DEMONSTRATES BASIC'S ABILITIES TO PEEK,"
41 PRINT"      ACCEPT HEX INPUT, AND PERFORM LOGICAL OPERATIONS"
50 SET 0,80
60 DIM Q$(100)
70 INTEGER A,I,C
80 DIM A$(15) ; A$="0123456789ABCDEF"
90 PRINT
100 REM
110 REM GET USER'S STARTING ADDRESS
120 REM
125 ON ESC GOTO 9000
127 @ ; @ ; @
130 Q$="%" : INPUT"ENTER STARTING ADDRESS FOR DUMP (IN HEX) >>",Q$(1)
135 ON ESC GOTO 100
140 Q$(LEN(Q$))="%"
150 I=VAL(Q$)
160 @ ; @ ; @
190 C=0
200 REM
210 REM PRINT HEADING FOR NEXT LINE
220 REM
230 A=BINAND(I,%FF00%)
240 A=BINAND(%00FF%,A/%0100%)
250 GOSUB 900
260 A=BINAND(I,%00FF%)
270 GOSUB 900
280 PRINT"    ";
300 REM
310 REM PRINT LINE OF HEX DATA
320 REM
330  FOR J=1 TO 16
340  A=PEEK(I) : I=I+1
350  GOSUB 900
360  PRINT" ";
380  NEXT J
390 I=I-16 : PRINT"    ";
400 REM
410 REM PRINT SAME LINE AS ASCII CHARACTERS
420 REM
430  FOR J=1 TO 16
440  A=BINAND(PEEK(I),%007F%) ; I=I+1
450  Q$=".." ; IF A>%001F%THEN IF A<%007F%THEN Q$=CHR$(A)
460  PRINT Q$(0,0);
470  NEXT J
480 PRINT
500 REM
510 REM SPACING CONTROL
520 REM
530 C=C+1 : IF BINAND(C,%0007%)=0 THEN PRINT
590 GOTO 200
900 REM
910 REM THIS SUBROUTINE PRINTS TWO HEX DIGITS FROM CONTENTS OF A
920 REM
930 PRINT A$(BINAND(A,%00F0%)/%0010%,-1);A$(BINAND(A,%000F%),-1);
990 RETURN
9000 REM
9010 REM COME HERE ON ESC FROM INPUT STATEMENT
9020 REM
9030 END
```

APPENDIX A: BASIC ERROR MESSAGES

FATAL ERRORS

STAND ALONE BASIC ERROR CODE	DISK BASIC ERROR NUMBER	MESSAGE	MEANING
SY	1	Syntax	This error covers a multitude of errors which can occur when the user is entering (typing in) a program. For example: Unmatched parentheses: A=(B*(C) Misspelled words: PIRNT A Wrong data type: A$=3*A Bad punctuation: PRINT A(7;2) etc. Because there is only one message for all these errors, a dollar sign is printed under the line in error at a position approximately indicating where the error was detected.
US	2	Using Syntax	The format string for a PRINT USING statement is in error. For example: PRINT USING"#.##!!!",3.2E9 (only 3 exclamation marks; 4 required)
#A	3	Number of Arguments	A function call requires a different number of arguments than the number passed to it. For example: DEF FNA(X,Y)=X+Y PRINT FNA(J)
NA	4	No Disk in System	In a stand-alone system, a request for a disk-only function was made. e.g., OPEN\1\"LP" (presumably the user meant "$LP")
IL	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 time-sharing system.

FATAL ERRORS (cont.)

STAND ALONE BASIC ERROR CODE	DISK BASIC ERROR NUMBER	MESSAGE	MEANING
PS	6	Print Item Size	An attempt was made to print a single item which required more characters than the current page width. e.g., SET Ø,1Ø PRINT "LOTS OF CHARACTERS"
#G	7	Too Many GOSUBs	Subroutines are nested within subroutines to a depth which exceeds that allowed by BASIC.
#(	8	EXP Too Complex	Too many levels of expressions, too many parentheses or function references.
RT	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 RETURN statement.
NX	10	NEXT Without FOR	FOR and NEXT statements must be paired. This error may occur if a line containing a FOR statement is deleted.
FU	12	Function not Defined	A user function is referenced by the program but has not been defined. If the line containing the DEF FNs(X) is deleted, the function is no longer defined.
DV	13	DIMension Statement Error	Invalid argument(s) in the DIMension statement. For example: negative number: DIM A (-20) too many subscripts: DIM B (5,5,5,5)
L#	14	GOTO/GOSUB Undefined Line Number	A GOTO or GOSUB statement refers to a line that does not exist.
SS	15	Subscript Values	The values assumed by subscripts must be less than those in the DIMension statement.

FATAL ERRORS (cont.)

STAND ALONE BASIC ERROR CODE	DISK BASIC ERROR NUMBER	MESSAGE	MEANING
#S	16	Number of Subscripts	The number of subscripts associated with a variable must match the number of subscripts in the DIMension statement.
EL	101	End of Statement/ End of Line	This is an internal BASIC error - please document and call CROMEMCO.
SZ	102	Array or String Space Overflow	There is not enough memory to store the array or string.

USER TRAPPABLE (NON-FATAL) ERRORS

STAND ALONE BASIC ERROR CODE	DISK BASIC ERROR NUMBER	MESSAGE	MEANING
NF	128	File Not Fnd	File not found on disk (file not in directory) or the device name is not in the device directory list.
FN	129	Filename	A disk file name was used in a Stand Alone System or an illegal file name was passed.
CM	130	Invalid Cmd 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.
FO	131	File Already Open	An OPEN command was given to a file which was already open.
NO	132	File Not Open	A read or write was attempted using a file which had not been OPENed.
F#	133	File Number	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.
OP	134	Cannot Open File	A message from the device driver (or CDOS). (A non-zero statement returned on OPEN.)
FS	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)
NA	136	File Mode Error	A READ was attempted from a write only file or vice versa.
NA	137	File Already Exists	An attempt was made to CREATE a file that already exists.

USER TRAPPABLE (NON-FATAL) ERRORS (cont.)

STAND ALONE BASIC ERROR CODE	DISK BASIC ERROR NUMBER	MESSAGE	MEANING
NA	138	File Read: No Data	End of file read, or, for random access only, an attempt to read a portion of the file which has not been written.
NA	139	File Write	A message from CDOS - an attempt was made to write to a protected disk.
NA	140	File Positon	An attempt was made to read a negative file record or a record larger than 240K bytes.
NC	141	No Channel Available	All I/O channels in use. (The maximum number of channels is system dependent.)
HX	200	Invalid Hex Number	Invalid hexadecimal number. Hexadecimal numbers must contain only the characters 0 through 9 and A through F.
BO	201	Integer Overflow	A value greater than 32767 was assigned to an integer variable.
FA	202	Function Arg Value	A function was called using an illegal argument. For example: SQR (-2).
IN	203	Invalid Input	An attempt was made to INPUT non-numeric data into a numeric variable.
IN	204	Input	An attempt was made to INPUT more items than were called for in the INPUT instruction.
DM	205	Not Dimensioned	A reference was made to a subscripted variable which had not been dimensioned.
ND	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.

USER TRAPPABLE (NON-FATAL) ERRORS (cont.)

STAND ALONE BASIC ERROR CODE	DISK BASIC ERROR NUMBER	MESSAGE	MEANING
DT	207	Data Type Mismatch	An attempt was made to assign a string value to a numeric variable or vice versa. For example: A$(1)=5.
E#	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.
LL	209	Line Length	A line longer than 132 characters was entered.
NA	210	Input Timeout	See the SET instruction for information about this error.
OV	250	Overflow/ Underflow	An operation produced a number outside of the range 9.99E+62 to 9.99E-65. For example: A=1/0. An integer variable which is assigned a value less than -32767 can also give this message.

APPENDIX B: ASCII CHARACTER CODES

Decimal	Character	Decimal	Character	Decimal	Character.
000	NUL	043	+	086	V
001	SOH	044	,	087	W
002	STX	045	-	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	[
006	ACK	049	1	092	\
007	BEL	050	2	093	]
008	BS	051	3	094	↑
009	HT	052	4	095	←
010	LF	053	5	096	`
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060	<	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
031	US	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	'	082	R	125	}
040	(	083	S	126	~
041	)	084	T	127	DEL
042	*	085	U		

LF = Line Feed FF = Form Feed CR = Carriage Return DEL = Rubout

APPENDIX C: ADDING DEVICE DRIVERS TO BASIC

Declaring Device Drivers

All I/O devices to be accessed from BASIC must be declared in the device driver list. This list starts at location DDLIST* and each entry is 8 bytes long. These entries are of the following form:

byte Ø	device name (ASCII) first letter second letter	byte 1
2	base address of device driver LSB MSB	3
4	may be used for device dependent information needed by driver	5
6	reserved	7

The device name (bytes Ø and 1) is used for finding the correct device to OPEN. For example, OPEN\2\"$LP" would instruct BASIC to search the device driver list for device name "LP" (the $ indicates a device instead of a disk file). The driver's starting address (see definition below) occupies the next two bytes. The following two bytes

---

* Actual addresses and equated value may vary depending on both revisions and which version of BASIC is being used.

are reserved for use by the driver and their function(s) are defined by the person programming the driver. NOTE: This device driver list can be placed in PROM. These two bytes are thus not intended as temporary storage areas.

A typical use of one of these bytes might be to hold the actual device address to be accessed. This would, for example, allow a general purpose TU-ART driver to drive several ports each dedicated to a different device.

The last two bytes of each table entry are reserved for future use. The table itself is terminated by the first nul driver name (Hex ØØ in the first character of the name).

Rules for Device Drivers

The actual device drivers must follow certain prescribed rules. In particular, the first 16 byte locations in the driver must contain address pointers to the various routines (some of which are required) in a BASIC driver. The 16 bytes define the addresses of 8 different routines as shown in the following table:

	(LSB)	(MSB)	
Byte Ø	address of OPEN routine		Byte 1
2	address of CLOSE routine		3
4	address of SET STATUS routine		5
6	address of GET STATUS routine		7
8	address of PUTC routine		9
10	address of GETC routine		11
12	reserved (should be Ø)		13
14	reserved (should be Ø)		15

If any of the routines noted above are not defined for a given driver, the corresponding address field should be set to ØØØØH. However, if a routine is (or must be -- see below) defined but does not do anything, the corresponding address must contain the address of an XOR A/RET instruction sequence.

A description of what must be accomplished by the various routines follows. Parameters which may (or must) be passed between BASIC and the driver routines are always passed in registers, or in the Extended File Control Block (EFCB).

Most of the information needed by the driver subroutines may be obtained from (or saved in) the Extended File Control Block (EFCB).

Register usage is covered in the description of each "command" routine below:

OPEN: This routine should perform any processing required to initialize the device. For example, a line printer driver would typically issue a form feed on open.

On Entry: (A') = no. of parameters (Ø,1,or 2)
(IY) = ADDR of EFCB
Ø, 1, or 2 parameters are passed to the routine in locations EFCBP1 and EFCBP2 of the EFCB.

On Return: A ≠ Ø says can't open

CLOSE: This routine should perform processing necessary to "shut down" the device. For example, a paper tape punch driver might punch out a trailer piece of tape.

On Entry: (IY) = ADDR of EFCB

On Return: A ≠ Ø says can't close

PUTC:
GETC: These routines perform byte-by-byte transfers to and from the device. Devices requiring a buffer may use the buffer (and/or extended buffer) in the EFCB.

On Entry: (IY) = ADDR of EFCB
(A) is character to be written (PUTC only)

On Return: (A) used for character read (GETC only)

SPOS: (SSTAT)
This routine is used to set the "status" of the device. For example, a DAZZLER driver might use this to set the X/Y screen position.

On Entry: (A') = no. of parameters (Ø,1,or 2)
(IY) = ADDR of EFCB
Ø, 1, or 2 parameters are passed to the routine in locations EFCBP3 and EFCBP4 of the EFCB.

On Return: A ≠ Ø says invalid status request

GPOS: (GSTAT)

On Entry: (IY) = ADDR of EFCB
(A') = which status is requested

On Exit: (HL) contains appropriate status value

NOTE: Registers IX and IY cannot be changed by any of these routines.

The EFCB referred to has the following format:

```
EFCB    :  DS 1    ;0=not in use, 1= in use                          } For use by
EFCBDA  :  DS 2    ;Device Driver Address, bytes 2&3 from DDLIST      } BASIC only
EFCBDD  :  DS 2    ;Device Dependent info, bytes 4&5 from DDLIST
EFCBP1  :  DS 2    ;\These are the two optional parameters
EFCBP2  :  DS 2    ;/P1 & P2 as used in the "OPEN" statement
EFCBP3  :  DS 2    ;\These are the 2 optional parameters P1 & P2
EFCBP4  :  DS 2    ;/used in PUT, GET, PRINT, and INPUT statements
BUFFER  :  DS 179  ;Available to user for accumulating individually
                   ;passed bytes into a buffer
```

When the proper DDLIST entries have been made as shown, operation of the driver is as follows:

OPEN\1,A,B\"$DR"	Calls the OPEN routine of driver "DR". EFCBP1 = A, EFCBP2 = B A' register = 2, IY= ADDR of EFCB If OPEN is not needed, it must consist of an XOR A, RET sequence.
PRINT\1,X,Y\"HELLO"	The Set Status routine of "DR" is called with EFCBP3 = X, EFCBP4 = Y (X and Y are converted to integers first). A' = 2, IY = ADDR of EFCB An XOR A, RET sequence should be executed at the completion of Set Status. Next, the PUTC routine of "DR" is called 7 times, once for each character to be transmitted plus once with a Carriage Return and once with a line feed. A = character, IY = ADDR of EFCB
INPUT\1\A$	Set Status routine of "DR" is called with EFCBP3 = %FFFF%, EFCBP4 = %FFFF%, A' = 0, IY = ADDR of EFCB An XOR A, RET sequence should be executed. Next, the GETC routine of "DR" is called repeatedly, expecting one byte to be returned in A. This continues until a terminator (CR,LF,FF,NULL) is transmitted, or until more than 132 characters have been transmitted.
A = IOSTAT(1,n) where 0=n=255	The Get Status routine of "DR" is called. A' contains n, the requested status parameter and IY = ADDR of EFCB. The status value to be returned should be placed in HL. An XOR A, RET sequence should be executed last.
CLOSE\1\	The CLOSE routine of "DR" is called. If CLOSE is not needed, an XOR A, RET sequence must be provided.

Note that if PUT and GET were used in the above calls instead of PRINT and INPUT, binary bytes would be transmitted according to variable type and no carriage control information would be sent. I.E., PUT\1\V would transmit two bytes if V is an integer, four bytes if V is short floating point, and eight bytes if V is long floating point. No carriage return or line feed is sent. Also note that PRINT\1\"A","B" will not transmit the comma but will send the proper number of spaces to place "B" in the next tab field.

```
ADDR  CODE
                ; NOTE: THE ADDRESSES SHOWN IN THIS LISTING MAY VARY FROM VERSION
                ; TO VERSION, but the format is identical.  The actual starting
                ; address may be found in locations 0412, 0413 for CDOS BASIC
                ;                                  and 8012, 8013 for Stand Alone BASIC
                DDLIST:
4B6C  53        DDCNSL  DB      'SY'            ;'SYSTEM' = CONSOLE
4B6D  59
4B6E  E84A              DW      DRTUART
4B70  00                DB      0,0
4B71  00
4B72  00                DB      0,0
4B73  00
4B74  54                DB      'T5'            ;2ND TUART PORT
4B75  35
4B76  E84A              DW      DRTUART
4B78  50                DB      50H,50H         ;BOTH BYTES GET ADDR OF 2ND TUART
4B79  50
4B7A  0000              DW      0               ;RESERVED BY SYSTEM
                ;
                ; CDOS SYSTEM PUNCH, READER, AND LIST DRIVERS
4B7C  50                DB      'PU'
4B7D  55
4B7E  9C4A              DW      DRCDPU          ;CDOS PUNCH DRIVER
4B80  0000              DW      0
4B82  0000              DW      0
4B84  52                DB      'RD'
4B85  44
4B86  B54A              DW      DRCDRD          ;CDOS READER
4B88  0000              DW      0
4B8A  0000              DW      0
4B8C  4C                DB      'LP'
4B8D  50
4B8E  CA4A              DW      DRCDLP          ;CDOS LIST DRIVER
4B90  0000              DW      0
4B92  0000              DW      0
4B94  00        DDEND:  DB      0               ;END OF DDLIST-NOTE: IF DDLIST IS
                ;                               EXTENDED, THIS NULL  should be
                ;                               overwritten and then placed at end
                ;                               of new DDLIST
      4B95              DEFS    64              ;RESERVE SPACE FOR MORE DRIVERS
      FFFF               IF     SOSVER          ;--------------------------
                        DW      !DUMMY
                         ELSE                   ;----------------------------------
```

```
ADDR  CODE      ; SAMPLE DEVICE DRIVERS
                ;
                ; PUNCH
                ;
                DRCDPU:
4A9C  E64A              DW      !DUMMY
4A9E  E64A              DW      !DUMMY
4AA0  E64A              DW      !DUMMY
4AA2  E64A              DW      !DUMMY
4AA4  AC4A              DW      !PUPC           ;PUNCH PUT CHARACTER
4AA6  0000              DW      0
4AA8  0000              DW      0
4AAA  0000              DW      0
                ;
4AAC  5F        !PUPC:  LD      E,A             ;GET CHAR TO E REG AS CDOS EXPECTS
4AAD  0E04              LD      C,4             ;CDOS PUNCH ENTRY
                !PULPJ:
4AAF  F5                PUSH    AF
4AB0  CD0500            CALL    CDOS
4AB3  F1                POP     AF
4AB4  C9                RET
                ;
                ;
                DRCDRD:
4AB5  E64A              DW      !DUMMY
4AB7  E64A              DW      !DUMMY
4AB9  E64A              DW      !DUMMY
4ABB  E64A              DW      !DUMMY
4ABD  0000              DW      0
4ABF  C54A              DW      !RDGC           ;RDR GET CHAR ROUTINE
4AC1  0000              DW      0
4AC3  0000              DW      0
                ;
                !RDGC:
4AC5  0E03              LD      C,3             ;CDOS READER GETC ENTRY PARM
4AC7  C30500            JP      CDOS            ;READY TO GO... CHAR RTND IN A
                DRCDLP:
4ACA  DF4A              DW      !LPOP           ;OPEN ROUTINE
4ACC  E64A              DW      !DUMMY
4ACE  E64A              DW      !DUMMY
4AD0  E64A              DW      !DUMMY
4AD2  DA4A              DW      !LPPC           ;AND LP PUTC ROUTINE
4AD4  0000              DW      0
4AD6  0000              DW      0
4AD8  0000              DW      0
                !LPPC:
4ADA  5F                LD      E,A             ;MOVE CHAR TO E
4ADB  0E05              LD      C,5             ;CDOS LIST WRITE ENTRY PARM
4ADD  18D0              JR      !PULPJ
                !LPOP:
4ADF  3E0C              LD      A,ASCFF         ;ISSUE A FORM FEED ON OPENING FILE
4AE1  CDDA4A            CALL    !LPPC
4AE4  AF                XOR     A
4AE5  C9                RET                     ;RET WITH 0 STATUS
                !DUMMY:
4AE6  AF                XOR     A
4AE7  C9                RET
```

TU-ART I/O PORT DRIVER

Note that there is a general purpose TU-ART serial I/O port driver in the DDLIST called "T5". It assumes a TU-ART addressed at port 50H and may be used for any serial device provided the baud rate is initialized prior to its use with an "OUT" statement.

Example:

```
10  OUT %50%,4 :  REM SET TU-ART TO 300 BAUD
20  OPEN\2\"$T5"  :  REM SETS UP DEVICE #2 FOR SERIAL I/O
```

The driver for the system console at port Ø is labeled "$SY". To switch between the console and a second terminal:

```
20  OPEN\1\"$SY"
30  OPEN\2\"$T5"
```

This will allow you to later say PRINT\A\"MESSAGE", where A can be changed via software to be 1 for the console and 2 for another terminal.

See also p 175/6

$PU - punch
$RD paper tape reader
$LP line printer

Baud rate	One Stop bit	two Stop bit
100	81	01
150	82	02
300	84	04
1200	88	08
2400	90	10
4800	A0	20
9600	C0	40

CHANGING THE NUMBER OF I/O CHANNELS

BASIC carries 4 I/O channels in addition to the console. One I/O channel is needed for each file which is OPENed at the same time. Each I/O channel occupies 192 bytes of memory.

If you wish to change the number of channels in use, type the following commands while in BASIC:

Disk System	Stand Alone System
POKE%416%,X	POKE%8016%,X
A=USR(%400%,0)	A=USR(%8000%,0)

where X is the number of I/O channels desired in addition to the console. The POKE command puts the number of channels desired into the proper location and the USR function re-initializes BASIC.

On a disk system, you can now save this new version of BASIC by returning to CDOS and saving it:

```
>>BYE
A. SAVE BASIC1.COM 76
```

APPENDIX D:

CROMEMCO 16K Z-80 EXTENDED BASIC AREAS OF USER INTEREST

Note: 1. In the address field, XX stands for: 04 if you are using CDOS
80 if you are using SOS

2. CDOS is CROMEMCO DISK OPERATING SYSTEM
SOS is STAND ALONE OPERATING SYSTEM

Hex Address	No. of Bytes	Version	Description
XXØØ	3	CDOS	A jump (JP) to the routine where BASIC uses the address of the bottom of CDOS to establish the top of user program space.
		SOS	A jump (JP) to the routine which uses repeated RETURN characters to initialize the band rate. After baud rate is established, BASIC automatically "sizes" RAM memory (starting at location Ø) to establish top of user program space.
XXØ3	3	Both	A jump (JP) to a point in BASIC equivalent to a request to enter a new program line. Does not destroy program currently in memory.
XXØ6	3	Both	A jump (JP) to a point in BASIC equivalent to issuing a SCRatch command. Does not re-size memory.
XXØ9	3	Both	A jump (JP) to a point in BASIC equivalent to the point reached via the jump at XXØØ except that memory sizing and baud rate initialization do not take place. On entry here, the HL register pair should contain the address to be used as top of user program space, and the A register should contain the number of file channels to allocate.
XXØC	3	CDOS	An illegal instruction (FF Hex, actually a RST 38H), which CDOS traps and uses to display an error message.
		SOS	A jump (JP) to the "IL" (Illegal command) error message.

CROMEMCO 16K Z-80 EXTENDED BASIC AREAS OF USER INTEREST

Hex Address	No. of Bytes	Version	Description
		Both	If a CROMEMCO monitor resides in PROM at EØØØ Hex, these 3 bytes should be changed to a jump (JP) to the breakpoint handler in the monitor. In case of failure in BASIC, the breakpoint information thus displayed might prove extremely valuable.
XXØF	3	CDOS	A jump (JP) to the CDOS warm-start at address ØØØØ. Used by the 'BYE' command to return to CDOS.
		SOS	A jump (JP) to location EØØ8 Hex, the warmstart point in the CROMEMCO monitor. Used by the 'BYE' command to exit to the monitor. CAUTION: 'BYE' should not be used in systems without a monitor.
XX12	2	Both	The address pointer used by BASIC to locate the beginning of the Device Driver LIST. The user may use this address to find the DDLIST or may change it to force BASIC to use an alternate DDLIST.
XX14	2	CDOS	The address pointer to the beginning of the ERROR MESSAGE. The format of error messages in memory is as follows: 3 Bytes -- Call to error printer 1 Byte -- Error number n Bytes -- ASCII message, terminated by a ØØ byte. 3 Bytes -- Call to error-printer 2 Bytes -- The error code, second byte first.
XX14	2	SOS	The address pointer to the beginning of the ERROR MESSAGES. The format of error messages in memory is as follows: 3 Bytes -- Call to error printer 2 Bytes -- The error code, second byte first.

CROMEMCO 16K Z-80 EXTENDED BASIC AREAS OF USER INTEREST

Hex Address	No. of Bytes	Version	Description
XX16	1	B	The maximum number of channels which can be used. The default is 4.
XX17	1	B	The number of characters per line. The default value is 80. See SET instruction.
XX18	1	B	The number of characters per TAB position. The default value is 20. See the SET instruction.
XX19	1	B	The character which is used as a rubout, default value is 5FH (underline).
XX1A	1	Both	Type of numeric variables and constants where 1=integer, 2=short floating point, and 4=long floating point. The default value is 4.
XX1B	1	Both	Line delete character. The default value is control-U.
XX1C	1	Both	The carriage return delay (number of nul characters sent after a carriage return). The default value is 0.
XX1D	4	Both	Backspace echo - must terminate with a 0. The default value is backspace, space, backspace, 0.

PATCH SPACE

Disk Version: The only space available in the disk version of 16K BASIC for user patches is at locations 425 to 4FF and 103 to 1FF. The user can put and SAVE (CDOS) a patched file at these locations.

APPENDIX E:

PROGRAMMING HINTS

1) A$(-1) = "?" + A$(-1) is a handy way to fill a string with any character (a question mark in this example).

2) Programs execute significantly faster if variables used in loops and/or for subscripting are declared as type INTEGER.

3) SAVEd programs may or may not be compatible between various revisions of 16K BASIC. LISTed programs always are!

4) In Cromemco 16K BASIC (unlike most microcomputer BASICs) constants used in program lines cause execution speeds as fast or faster than variable references. In addition, integer constants occupy no more room than a variable reference.

5) Using the Øth element of arrays (and strings) can save memory space.

6) BASIC may be entered from CDOS and requested to immediately RUN any SAVEd program by entering:

BASIC <program name>

at the CDOS command processor's prompt.

7) When using ENTER to overlay portions of running programs, like line numbers in various overlays will cause most efficient memory usage.

8) Although files OPENed for READ/WRITE may be used as WRITE-ONLY files, slightly faster execution speeds may occur if the file is OPENed as WRITE-ONLY.

9) Use SET Ø,-1 to disable page width checking. This is especially useful for graphics output to Diablo-type printers.

10) VAL will return a zero value if the first character of the argument string is not a number. This can be an extremely useful way of decoding a user's input.

PROGRAMMING HINTS (cont'd)

11) To EXECUTE a BATCH command from BASIC, PRINT the desired commands to a file called "A:BASIC.CMD" and then execute the following program:

```
100  CREATE "A:$$$$.CMD"
200  OPEN\1\"A:$$$$.CMD"
300  PUT\1\CHR$(11),"@ BASIC.CMD$"
400  CLOSE
500  DSK "A:"
600  BYE
```

You must specify the appropriate disk drive because "$$$$.CMD" appears as a peripheral device to BASIC. "B:" could have been used throughout the program above instead.

MEMORY REQUIREMENTS

Your Cromemco 16K Disk BASIC is designed to run in a system with 32K of contiguous memory beginning at memory location zero. This memory should use two Cromemco 16KZ RAM cards or eight Cromemco 4KZ RAM cards.

For long BASIC programs you may wish to use 48K of memory instead of 32K. In order to allow you to do this, a version of CDOS designed to run in 48K is available as a file on your BASIC diskette. For 5" diskettes this file is named DOSMIN48.SYS. For 8" diskettes this file is named DOSMAX48.SYS.

If you do have 48K of memory in your system and wish to make use of all this memory for writing long BASIC programs, you can use the CDOS WRTSYS command to replace the 32K CDOS system with a 48K system. for the 5" diskette the form of this command is:

```
WRTSYS A:=DOSMIN48.SYS
```

For the 8" diskette the form of this command is:

```
WRTSYS A:=DOSMAX48.SYS
```

CHAINING BASIC PROGRAMS:

These programs demonstrate the ability of Cromemco's 16K Extended BASIC to chain or have one program call 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 chained together. In the following example, program ONE is run and calls program TWO.

```
PROGRAM "ONE"

 1 GOTO 10
 5 ENTER"TWO"
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"

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"

>>RUN

10              35              45

***50 END***
```

When program ONE is entered and RUN, control is passed to line 1Ø by line 1. Then at line 2Ø, the DATA from line 1Ø is READ into variables A through I. Control is then passed to line 5 which calls in the second program. Program TWO overlays lines 1Ø, 2Ø, and 3Ø. The line which follows line 5 is line 1Ø. Now, however, line 1Ø is a line from program TWO. The rest of program TWO is executed with the results being printed out by line 4Ø.

Stop Dnrec OUT 52, 1 (→30)

Start Dnrec OUT 52, 33.