COMPUCOLOR II

PROGRAMMING AND REFERENCE

MANUAL

999209 Rev. 2

TABLE OF CONTENTS

5. FUNCTIONS AND SUBROUTINES

6. ARRAYS

7. FURTHER SOPHISTICATION

APPENDICES

1. INTRODUCTION

1.1 The COMPUCOLOR II

The COMPUCOLOR II will gladly introduce itself with but the slightest help from the user. Its brilliant colors and amazing versatility are easy to get to know. Once plugged in, it is ready to perform a myriad of tasks, both simple and complex. The user can easily insert a disk from the COMPUCOLOR library and have at his fingertips an assortment of games, recipes, financial statements and more. But for the more adventurous, (and COMPUCOLOR makes it fun to be adventurous!) COMPUCOLOR II offers the opportunity for the user to write his own programs. The language of communication for the COMPUCOLOR II is BASIC, a popular computer language developed at Dartmouth University to make programming easy for everyone.

BASIC is a single user, conversational programming language which uses simple statements and familiar mathematical notations to perform operations. BASIC is one of the simplest computer languages to learn, and once learned provides the facility found in more advanced techniques to perform intricate manipulations and express problems efficiently.

Like any other language, BASIC has a prescribed grammar to which the user must adhere in order to produce statements and commands intelligible to the computer. The following pages provide a quick but complete introduction to the BASIC language and the features of the COMPUCOLOR II. Careful reading and liberal experimentation with examples will enable a user to start programming in a short time. Adopting a leisurely pace with the text will ensure that the new user will find programming much easier than suspected.

1.2 Initializing and Running BASIC

When the COMPUCOLOR II is turned on, the screen display for Model 3 will be:

```
DISK BASIC 8001 V.6.78 COPYRIGHT (C) BY COMPUCOLOR
MAXIMUM RAM AVAILABLE?
7473 BYTES FREE
READY
```

The number of free bytes on Models 4 and 5 will be 15665 and 32049, respectively. The READY message indicates that the machine is now ready to accept any BASIC programming statements that the user wishes to enter. If the user wishes to use a prepared program from one of the COMPUCOLOR II diskettes, the diskette must be slid into the opening on the right hand side of the machine, and the door must be closed. Pushing the AUTO key (the brown key on the upper left of the keyboard) will result in a list or "MENU" of available programs on the screen. A choice is indicated by typing in the number of the selected program.

The program will be loaded and the COMPUCOLOR II will proceed with instructions on how to use the program.

If, when the machine is powered on, the proper message does not appear, the user should hold the shift and control keys down while striking the CPU RESET key. This should produce the correct screen display, however, there may be a delay of 5 or more seconds before it appears. On the deluxe or extended keyboards the COMMAND key can be struck in place of the combined CONTROL SHIFT sequence.

It may often be necessary to reset BASIC after the machine has been turned on and a program or two has been run. The first step to reinitializing BASIC is striking the CPU RESET key. The screen will output:

```
COMPUCOLOR II CRT MODE V.6.78
```

Then, the ESC and W(BASIC) keys are hit in sequence. The machine will print the message:

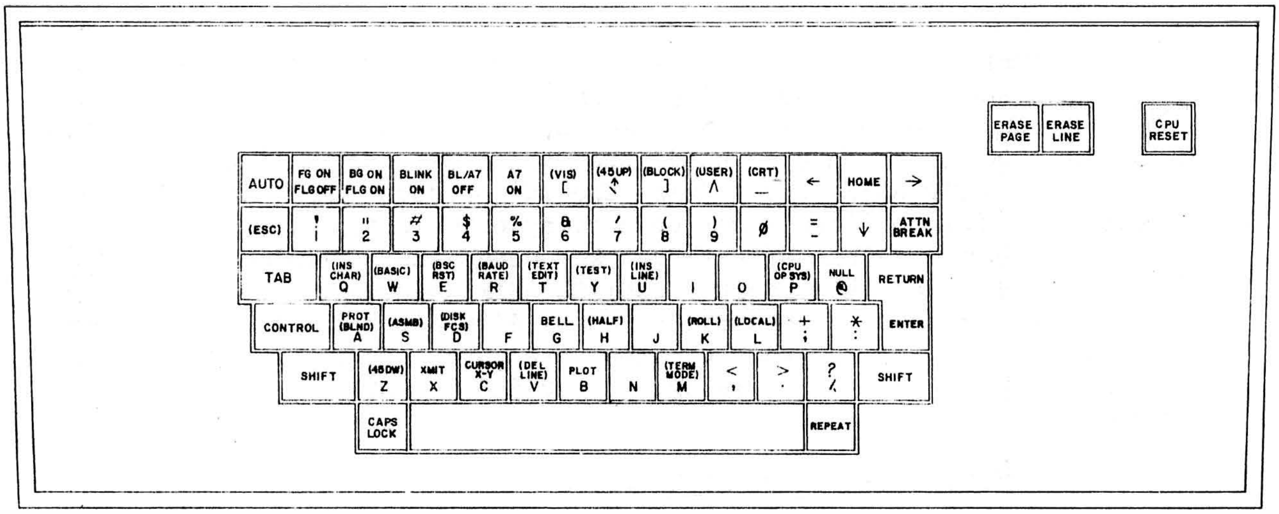
```
DISK BASIC 8001 V.6.78 COPYRIGHT (C) BY COMPUCOLOR
MAXIMUM RAM AVAILABLE?
```

If the user desires no specific amount of memory, then simply striking the RETURN key will bring the READY message to the screen. If, however, a certain amount of memory needs to be specified (as is necessary in some applications), the user must type in a number up to 8192, (or 16384 if the machine is a Model 4; or 32768 for Model 5) subtracting from this maximum any amount of space to be reserved as not for use by BASIC. The user then strikes the RETURN key and the machine will return the number of free bytes and the READY message. The machine is now ready, as when it is first turned on, to either accept a user's program or load a COMPUCOLOR II program from an inserted diskette. If it is necessary to leave BASIC by using the CPU RESET key or by entering the File Control System, BASIC can be re-entered without losing the program in its workspace by typing ESC and E (BST RST).

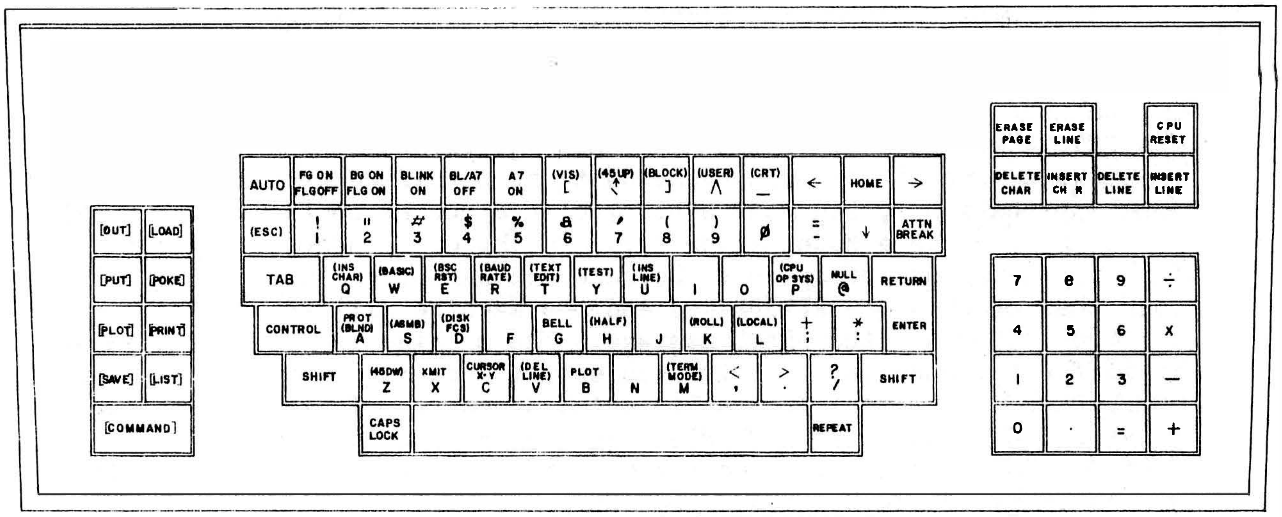
If the machine will not return the proper messages and/or numbers, the local dealer should be contacted for assistance.

1.3 COMPUCOLOR II Keyboard Layouts

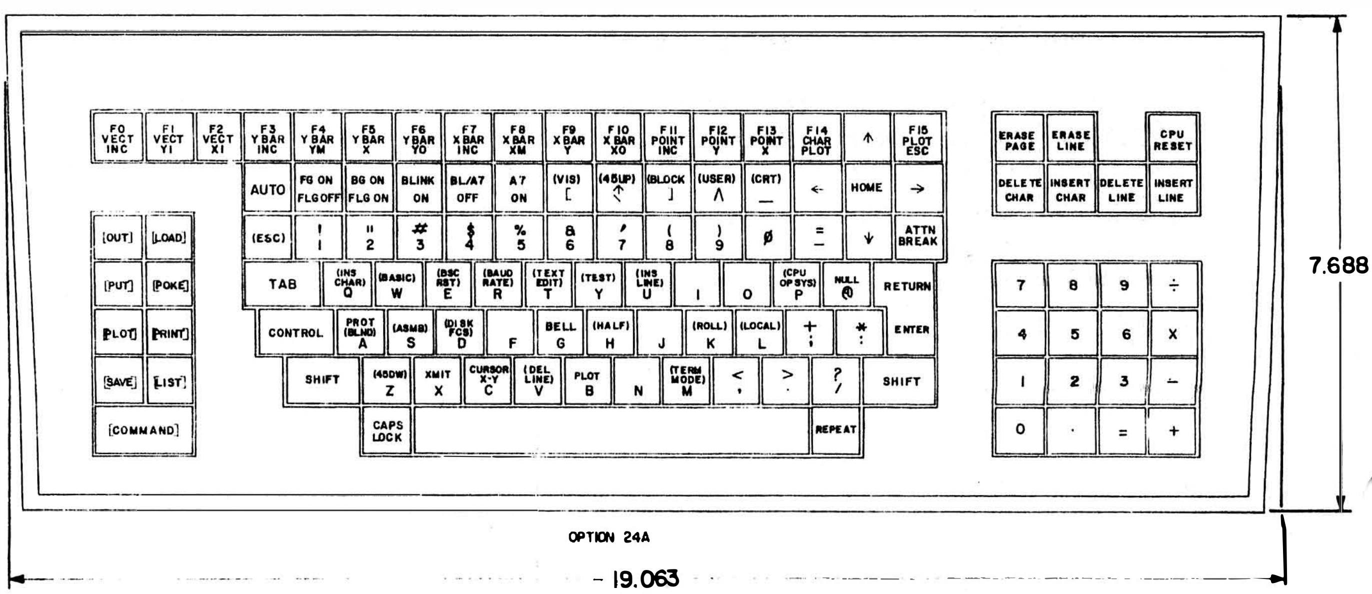
The layouts of the standard, extended, and deluxe keyboards are shown in the following figures.



STANDARD



OPTION 22A



OPTION 24A

1.4 Using the Manual

BASIC has thirty (30) key word program, editing, and command statements, eighteen (18) mathematical functions, nine (9) string functions and thirty (30) two-letter error messages. These features are described in detail in the next chapters, thus providing a ready reference to BASIC's capabilities. If the user is unfamiliar with the BASIC language, then the remaining portion of this manual should be studied in sequence while having a COMPUCOLOR II available to run the examples given.

Compucolor Corporation has a number of BASIC programs on the COMPUCOLOR II diskettes that are available at nominal prices. In addition, Compucolor will pay for BASIC programs that are provided on diskettes when properly documented and accepted for release on future Compucolor diskettes. Enjoy programming in BASIC!

2. ESSENTIALS FOR SIMPLE PROGRAMMING

2.1 Variables

BASIC uses variables as a basis for conveying values in programming statements. The variable is an algebraic symbol representing a number which the user assigns to it. A variable is formed in one of three ways. It can be a letter alone, a letter followed by a number, or two letters. For example:

Acceptable Variables	Unacceptable Variables
A	3F - begins with a digit
C2	25 - numeric constant
XY	
Q	

A variable longer than 2 characters will be accepted by BASIC, but BASIC will only read the first two characters. Thus, these must be distinct from any other variables used in the program. For example, CAT is not a new variable in a program already using the variable CANCEL. Words used as specific commands or statements in BASIC are reserved, and cannot be used as variable names (e.g. LIST, RUN, READ, etc.). If such a word is used, BASIC will not accept it as a variable, and will usually return an error message. Certain other special purpose variables are acceptable in BASIC, and will be described in later sections of this manual.

When the user assigns a value to a variable, it will retain that value until it is changed by a later statement or calculation in the program. All numeric variables, until given a value by the user, are assumed by the computer to have the value 0. String variables are initially assumed to be equal to the null string (see Section 3.10.) This assures that later changes or additions will not misinterpret values.

2.2 Numbers

BASIC treats all numbers (real and integer) as decimal numbers, that is, it accepts any decimal number and assumes a decimal point after an integer. The advantage of treating all numbers as decimal numbers is that any number or symbol can be used in any mathematical expression without regard to its type. Numbers used must be in the approximate range $10^{-38} < N < 10^{+38}$.

In addition to integers and real numbers, a third format for numbers is recognized and accepted by BASIC. This is the scientific or "E-type" notation, and in this format a number is expressed as a decimal number times some power of 10. The form is:

```
xxEn
```

where E represents "times 10 to the power of"; thus the number is read, "xx times 10 to the power of n." For example:

```
25.8E2 = 25.8 * 100  = 2580
```

Data may be input in any one or all three of these forms. Results of computations are output as decimals if they are within the range .01<n<999999; otherwise, they are output in E format. BASIC handles seven significant digits in normal operation and prints 6 decimal digits as illustrated below:

Value Typed In	Value Output by BASIC
.01	.01
.0099	9.9E-03
999999	999999
1000000	1E+06

BASIC automatically suppresses the printing of leading and trailing zeroes in integer and decimal numbers, and, as can be seen from the preceding examples, formats all floating point numbers in the form:

```
(sign) x.xxxxxE (+ or -)n
```

where x represents the number carried to six decimal places; E stands for "times 10 to the power of"; and n represents the value of the exponent. For example:

```
-3.47021E+08     is equal to     -347,021,000
7.26E-04         is equal to     .00726
```

Floating point format is used when storing and calculating most numbers. NOTE: Because memory size limitations prohibit the storage of infinite binary numbers, some numbers cannot be expressed exactly in BASIC. Accuracy is approximately 7.1 digits, and errors in the 6th digit can occur. For example; .999998 may be the result of some functions instead of 1. Discrepancies of this type are magnified when such a number is used in mathematical operations.

2.3 Arithmetic Operations

BASIC performs addition, subtraction, multiplication, division and exponentiation. Formulas to be evaluated are represented in a format similar to standard mathematical notation. The five operators used in writing most formulas are:

Symbol Operator	Example	Meaning
+	X+Y	Add Y to X
-	X-Y	Subtract Y from X
*	X*Y	Multiply X by Y
/	X/Y	Divide X by Y
^	X^Y	Raise X to Yth power

BASIC also permits the use of unary plus and minus. The - in -A+B, or the + in +X-Y are examples of such usage. Unary plus is ignored, while unary minus is treated as a zero minus the variable. The expression -A+B is processed as 0-A+B.

2.3.1 Priority of Arithmetic Operations

When more than one operation is to be performed in a single formula, as is most often the case, certain rules must be observed as to the precedence of operators. In any given mathematical formula, BASIC performs the arithmetic operations in the following order of evaluation:

1. Parentheses receive top priority. Any expression within parentheses is evaluated before an unparenthesized expression

2. Exponentiation

3. Unary minus

4. Multiplication and division (of equal priority)

5. Addition and Subtraction (of equal priority)

6. Logical operators in the order NOT, AND, then OR. (see Section 4.7)

If the rules above do not clearly designate the order of priority, then the evaluation of the expression proceeds from left to right. The expression A^B^C is evaluated from left to right as follows:

```
1.  A^B                    = step 1
2.  (result of step 1)^C   = answer
```

The expression A/B*C is also evaluated from left to right since multiplication and division are of equal priority:

```
1.  A/B                    = step 1
2.  (result of step 1)*C   = answer
```

The expression A+B*C^D is evaluated as:

```
1.  C^D                     = step 1
2.  (result of step 1)*B    = step 2
3.  (result of step 2)+A    = answer
```

Parentheses may be nested, or enclosed by a second set (or more) of parentheses. In this case, the expression within the innermost parentheses is evaluated first, and then the next innermost, and so on, until all have been evaluated. In the following example:

```
A = 7 * ((B^2+4) / X)
```

the order of evaluation is:

```
1.  B^2                     = step 1
2.  (result of step 1)+4    = step 2
3.  (result of step 2)/X    = step 3
4.  (result of step 3)*7    = A
```

Parentheses also prevent any confusion or doubt as to how the expression is evaluated. For example:

```
A*B^2/7+B/C*D^2                  ((A*B^2)/7)+((B/C)*D^2)
```

Both of these formulas are executed in the same way, but the order of evaluaton in the second is made more clear by the use of parentheses.

Spaces may be used in a similar manner. Since the BASIC interpreter ignores spaces (except when enclosed in quotation marks), the two statements:

```
B = D^2 + 1                      B=D^2+1
```

are identical in meaning and consequence, but spaces in the first statement provide ease in reading when the line is entered. When such a statement is subsequently printed by the computer, spaces entered on input are ignored, and the spacing will appear differently on the screen.

2.4 The Assignment Statement

The user assigns a value to a variable by the use of the equals (=) sign. The variable must appear on the left of the statement and its value on the right. For example:

```
A = 2
Q4 = 7.5
```

The statements 2=A , and 7.5=Q4 , although algebraically equivalent to the above examples, are not legal in BASIC, because the machine always takes the value on the right of the equals sign and assigns it to the variable on the left of the sign. The number 2 is not an acceptable variable, and the machine cannot replace its value with that of "A". The fundamental difference in meaning and use of the equals sign in algebra and in BASIC must be clearly understood to avoid confusion. In algebraic notation, the formula X=X+1 is meaningless. However, in BASIC (and in most other computer languages), the equals sign designates replacement rather than equality. Thus, this formula is actually translated: "add one to the current value of X and store the new result back in the same variable X." Whatever value has previously been assigned to X will be combined with the value 1. An expression such as A=B+C instructs the computer to add the values of B and C and store the result in a third variable A. The variable A is not being evaluated in terms of any previously assigned value, but only in terms of B and C. Therefore, if A has been assigned any value prior to its use in this statement, the old value is lost; it is instead replaced by the value B+C. For example:

X=2	Assigns the value 2 to the variable X.
X=X+1+Y	Adds 1 to the current value of X, then adds the value of Y to the result and assigns that value to X.

3. BEGINNING TO PROGRAM

3.1 Sample Program

The lines below form an acceptable BASIC program which the machine will understand and compute. The numbers at the start of each line are an essential part of the program. Each statement must have a line number in order to be executed when the program runs on the machine. The computer will process each line in ascending numerical order, regardless of the order in which it is typed into the machine.

```
10 A=8
20 B=7
30 C=A+B
40 PRINT C
```

The line number itself may be any integer from 0 to 65529, and lines may be numbered in increments as low as 1, but it is a good programming practice to number program statements in increments of 10 or 100. This leaves adequate room for insertion of statements at a later time without the necessity of renumbering the entire program. Hitting the return key at the end of a numbered line automatically enters that line into the computer and stores it in memory.

3.2 The PRINT Statement

Line 40 of the above program is a PRINT statement. This statement is necessary in order to retrieve the calculation the machine has made. After line 30, the computer has solved the problem and assigned the value 15 to the variable C. Without the PRINT statement, however, it will simply store that information for future use, and it will not be visible to the user. The PRINT statement need not always give the value of a single variable; it may contain an expression. Therefore, in the preceding program, the same result would have appeared if the program had read:

```
10 A=8
20 B=7
30 PRINT A+B
```

Other examples of the use of expressions in PRINT statements are:

```
10 A=400
20 PRINT A*975
```

```
10 R = 5
20 P = 3.14159
30 PRINT P*R^2
```

The PRINT statement can also be used to print a message or string of characters, either alone, or together with the evaluation and printing of numeric values. Characters to be printed are enclosed in double quotation marks. For example:

```
10 PRINT "CLASSIFIED"
20 PRINT "INFORMATION"
```

gives:

```
CLASSIFIED
INFORMATION
```

and:

```
10 A=50
20 PRINT "THE NEXT NUMBER IS",A
```

gives:

```
THE NEXT NUMBER IS 50
```

When a character string is printed, only the characters between the quotes appear; no leading or trailing spaces are added. Leading and trailing spaces can be added within the quotation marks using the keyboard space bar; spaces appear in the printout exactly as they are typed within the quotation marks.

A convenient shortcut in DISK BASIC is the use of the question mark (?) in place of the word "PRINT" in any PRINT statement. For example:

10 ?A	is equal to	10 PRINT A
30 ?"MAGIC"	is equal to	30 PRINT "MAGIC"

When the program is listed by the machine, however, the question mark is replaced by the word PRINT. (For a more detailed description of the PRINT statement, see Section 7.1)

3.3 The RUN Command

Once a program has been properly written and entered into the computer, the use of the RUN command will cause it to be processed by the machine and return the result of the program. When the last program line is typed and entered, the user types RUN and hits RETURN. Because RUN is a command and not part of the actual program, it needs no line number. The machine will return the result and the message READY. The READY message indicates that the machine is prepared to accept further additions or changes to the program. For example:

Program	Machine Response
10 R=50	
20 T=50	
30 PRINT R*T	2500
RUN	READY

If the user desires to write a completely new program, the machine must be cleared of existing data by re-initializing BASIC. (See 1.2.)

3.4 Corrections

Corrections can be easily made while programming. If, while typing a line, the user makes a mistake, the ← can be used to delete the last character typed. The ← moves the cursor back one space at a time, and it can be struck repeatedly until the error is erased. The line is then retyped from that point on, or, if the rest of the original line was correct, the → can be used to restore that portion of the line removed by the ←.

If the line containing the error is already entered, a correction is made by retyping the line correctly, using the same line number. The computer will replace the faulty line with the one most recently typed.

If the user desires to delete an entire line from the program, entering that line number and hitting RETURN will remove it from the program. The line currently being entered can be deleted by typing the ERASE LINE key.

The ERASE PAGE key will clear the entire CRT screen, but it does not change or disturb any BASIC statements in any way. It is often used to obtain a blank workspace on the screen while programming.

3.5 The REM Statement

It is often desirable to insert notes and messages within a program. Such data as the name and purpose of the program, how it is used, how certain parts of the program work, and expected results at various points are useful things to have present in the program for ready reference by anyone using that program.

The REMARK or REM statement is used to insert remarks or comments into a program without these comments affecting execution. Remarks do, however, use memory in the user area which may be needed by an exceptionally long program.

The REM statement must be preceded by a line number. The message itself can contain any legal character on the keyboard, including some of the control characters. BASIC completely ignores anything in a line following the letters REM. Typical REM statements are shown below:

```
10 REM THIS PROGRAM COMPUTES THE
15 REM ROOTS OF A QUADRATIC EQUATION
```

3.6 The LIST Command

The user can see a listing of his program on the screen by typing LIST and hitting RETURN. Such a listing makes finding errors much easier, and facilitates additions and changes to the program. A portion of any program may be viewed by typing LIST followed by a line number. The screen will show a listing of that line and all following lines in the program. Because the machine will scroll the program very rapidly, the user may wish to stop the listing at some point for a closer look. Hitting the BREAK key will cause the scrolling to halt. Hitting the RETURN key will resume the listing. To stop the listing altogether, so that the user can edit or change the program, the LINEFEED key (↓) is struck. This will produce the message READY.

3.7 The END Statement

The optional END statement is of the form:

```
END
```

Upon executing an END statement, program execution is terminated and the READY message is printed. Program execution can be continued at the statement immediately following the END statement by entering a CONT command. For example, executing the following lines:

```
10 PRINT 1: END: PRINT 2
20 PRINT 3
```

gives the following response:

```
RUN
 1

READY

CONT
 2
 3

READY
```

In this fashion the END statement can be used to generate program breaks to facilitate debugging a program.

Program execution will also terminate automatically when the program runs out of statements. Note that in both cases currently open files are not closed.

3.8 The CONT Command

The CONT command is of the form:

```
CONT
```

This command is used to continue program execution at the next statement after a program break or error is detected. Execution can be restarted at a specific line number by using a GOTO statement instead of CONT.

A CN error message is printed if it is impossible to continue execution after a program break. This message will appear if no program exists or a new or corrected line was entered into the program.

3.9 Multiple Statement Lines

For convenience in programming, DISK BASIC allows the user to place more than one statement on a single numbered line. The general form is:

```
statement:statement: ... :statement
```

where 'statement' is any permissible BASIC statement. Any number of statements may be put together on one line, with the restriction that line length must not exceed 96 characters. The colon (:) denotes new statements and separates them from one another. The statements are executed in order from left to right.

The user must take note of a few statements whose use in multiple statement lines requires some caution.

Because BASIC ignores anything after REM, in the following statement:

```
A=50:B=25:C=4:REM THIS PROGRAM ADDS:PRINT A+B+C
```

the result of A+B+C will never be computed and printed.

Because GOTO causes an immediate and unconditional transfer of control, anything following GOTO in a multiple statement line will never be executed. DATA statements that appear after GOTO's will, however, be read by any corresponding READ statements.

Care must be taken when IF...THEN statements are used in multiple statement lines. If the result of the test is false, control will not pass to the next statement in the line, but rather to the next numbered statement. For example:

```
50 C=2:A=5:IF A=6 THEN PRINT 1:PRINT 2
60 IF C=2 THEN PRINT 3:PRINT 4
```

This program will print out the numbers 3 and 4. If the IF...THEN statement comparison is true and does not pass control to a specific line number, the next statement to the right in the multilple statement line will be executed. For example:

```
40 A=10: IF A=10 THEN B=500: PRINT A+B
```

will result in setting B to 500 and the printing of the result of A+B.

3.10 Introduction to Strings

The previous sections described the manipulation of numerical information only; however, DISK BASIC also processes information in the form of character strings. A string, in this context, is a sequence of characters treated as a unit. A string is composed of alphabetic, numeric, or special characters. The maximum length of quoted strings and strings entered using the INPUT statement is determined by the length of the input line buffer which is 96 characters or bytes.

Any variable name followed by a dollar sign ($) character indicates a string variable. For example:

```
A$
C7$
LONG$
```

are simple string variables and can be used as follows:

```
10 A$="HELLO"
20 PRINT A$
```

Note that the string variable A$ is separate and distinct from the variable A. In DISK BASIC, all control characters above control code C (or 3) are legal characters within quotes (") except for the following:

```
Control Code K or 11 or erase line
Control Code L or 12 or erase page
Control Code M or 13 or return/enter
Control Code Y or 25 or cursor right
Control Code Z or 26 or cursor left
```

Concatenation is a string operator that puts one string after another without any intervening characters. It is specified by a plus sign (+) and works only with strings. The maximum length of a concatenated string is 255 characters. In each of the following examples, D$ contains the result of concatenating the strings A$, B$, and C$.

```
10 A$ = "33"
20 B$ = "22"
30 C$ = "44"
40 D$ = A$+B$+C$
50 PRINT D$

RUN
332244
```

```
10 A$ = "I AM"
20 B$ = " A CLEVER"
30 C$ = " COMPUCOLOR II"
40 D$ = A$+B$+C$
50 PRINT D$

RUN
I AM A CLEVER COMPUCOLOR II
```

3.11 The CLEAR Statement

The CLEAR statement clears all the user's variables including simple variables and arrays. The CLEAR statement has two forms as shown below:

```
CLEAR
```

and

```
CLEAR expression
```

The difference between the two forms is that the form with the expression specifies the new number of bytes in the string space. Upon entry to BASIC the string space is initialized to 50 bytes. For example, in programs that heavily use strings, this allocation can be changed by executing a CLEAR 250; it should be one of the first executed statements in a program because it also clears all the variables. For further information on how strings are allocated in the string space, see Section 7.4.

3.12 Immediate Mode

It is not necessary to write a complete program to use BASIC. Most of the statements discussed in this manual can be included in a program for later execution or given as commands which are immediately executed by the DISK BASIC interpreter. This latter facility makes BASIC an extremely powerful calculator.

BASIC distinguishes between lines entered for later execution and those entered for immediate execution solely by the presence (or absence) of line numbers. Statements which begin with line numbers are stored as part of the program; statements without line numbers are executed immediately upon being entered into the system. Thus the line:

```
10 PRINT "THIS IS A COMPUCOLOR II"
```

produces no action at the console upon entry, while the statement:

```
PRINT "THIS IS A COMPUCOLOR II"
```

causes the immediate output:

```
THIS IS A COMPUCOLOR II
```

Multiple statements can be used on a single line in immediate mode. For example:

```
A=1:PRINT A     gives:          1
```

Program loops are also allowed in immediate mode; thus a table of squares can be produced as follows: (For a description of FOR NEXT loops, see Section 4.9)

```
FOR I=1 TO 10: PRINT I, I^2:NEXT I

 1         1
 2         4
 3         9
 4         16
 5         25
 6         36
 7         49
 8         64
 9         81
 10        100

READY
```

3.13 Samples and Examples

In order to become more adept at programming, any user previously unfamiliar with BASIC should set aside some time for experimentation with the information thus far provided in this manual. Simple programs such as the ones below make good practice efforts.

A

```
10 REM THIS PROGRAM COMPUTES
20 REM THE AREA OF A CIRCLE
30 REM THE FORMULA IS:
40 REM AREA = PI * RADIUS ^ 2
50 PI = 3.14159
60 R = 25
70 A = PI * R ^ 2
80 PRINT "AREA = ",A
```

B

```
10 REM THIS PROGRAM AVERAGES
20 REM FIVE NUMBERS
30 A=23
40 B=1
50 C=188
60 D=5
70 E=89
80 T=A+B+C+D+E
90 AV=T/5
95 PRINT "AVERAGE =",AV
```

Write programs to solve these problems:

A

How many cubic yards of soil can be into put into a box that measures 5 feet by 42.5 inches by 1 yard?

B

Convert 40 degrees Fahrenheit into degrees Celsius using the formula C = (5/9)*(F-32)

4. MORE STATEMENTS, COMMANDS, AND FEATURES

4.1 The INPUT Statement

The INPUT statement is used when data values are to be entered from the terminal keyboard during program execution. The form of the statement is:

```
INPUT list
```

where 'list' is a list of variable names separated by commas. For example:

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

causes the computer to pause during execution, print a question mark, and wait for the entry of three numeric values separated by commas. The values are input to the computer by typing the RETURN key.

If too few values are entered, BASIC prints another ? to indicate that more data values are needed. If too many values are used, the excess data values on that line are ignored, but the program will continue. The values entered in response to the INPUT statement cannot be continued on another line and are terminated by the RETURN key. Values must be separated by commas if more than one value is entered on the same line.

When reading numeric values, spaces are ignored. When a non-space is found, it is assumed to be part of a number; if not, then the question mark is repeated. The number is terminated by a comma, colon, or carriage return.

When reading string items, leading spaces are ignored. When a non-space character is found, it is assumed to be the start of a string item. If this first character is a quotation mark ("), the item is taken as being a quoted string and all characters between the first double quote (") and a matching double quote or carriage return are returned as characters in the string. Thus, quoted strings may contain any legal character except double quote. If the first non-space character is not a double quote, then it is assumed to be an unquoted string constant. The string will terminate with a comma, colon, or carriage return.

When there are several values to be entered via the INPUT statement, it is helpful to print a message explaining the data needed. For example:

```
10 PRINT "YOUR AGE IS"
20 INPUT A
```

The INPUT statement can also contain quoted strings. The above example could be written:

```
10 INPUT "YOUR AGE IS?";A
```

Note that when a quoted string is included in an INPUT statement, the normal ? is not printed as a prompt character, and if desired, must be included as shown within the quotes above.

The INPUT statement allows BASIC to be programmed to accept direct questions and answers as well as fill-in-the-blank applications.

If the user wishes to stop a program while it is waiting at an input statement, LINEFEED and RETURN must be typed in sequence. If RETURN is typed in response to the INPUT prompt (?), DISK BASIC will assume the value 0 for numeric variables, and "0" for string variables. If there are additional variables in the INPUT list, a question mark (?) will be printed as discussed above.

4.2 The DATA Statement

The DATA statement is used in conjunction with the READ statement to enter data into an executing program. One statement is never used without the other. The form of the statement is:

```
DATA value list
```

where value list contains the numbers or strings to be assigned to the variables listed in a READ statement. Individual items in the value list are separated by commas; strings are usually enclosed in quotation marks. For example:

```
150 DATA 4,7,2,3,"ABC"
170 DATA 1,34E-3,3,171311
```

The scanning of numeric and string items is identical to that described above in the INPUT statement. An SN error message can result from an improperly formatted DATA list.

The location of DATA statements is arbitrary as long as they appear in the correct order; however, it is good practice to collect all related DATA statements near each other.

When the RUN command is executed, BASIC searches for the first DATA statement and saves a pointer to its location. Each time a READ statement is encountered in the program, the next value in the DATA statement is assigned to the designated variable. If there are no more values in that DATA statement, BASIC looks for the next DATA statement.

4.3 The READ Statement

A READ statement is used to assign the values listed in the DATA statements to the specified variables. The READ statement is of the form:

```
READ variable list
```

The items in the variable list may be simple variable names or string variable names and are separated by commas. For example:

```
10 READ A, B$, C
20 DATA 12, "42", .12E2
```

Since data must be read before it can be used in a program, READ statements generally occur near the beginning of the program. A READ statement can be placed anywhere in a multiple statement line.

If there are no data values available in the DATA statements for the READ to store, the out of data message below is printed:

```
OD ERROR IN xxxxx
READY
```

Items in the data list in excess of those needed by the program's READ statements are ignored.

4.4 The RESTORE Statement

The RESTORE statement causes the program to reuse the data from the first DATA statement, or, if a line number is specified, from the first DATA statement on or after the specified line. The two forms of the RESTORE statement are as follows:

```
RESTORE
```

and

```
RESTORE line number
```

For example:

```
100 RESTORE 50
```

causes the next READ statement to start reading data from the first DATA statement on or after line 50. The following example shows how the RESTORE statement functions:

```
10 INPUT " ENTER 1 FOR NUMERIC, 2 FOR STRINGS:"; A
20 IF A = 2 THEN 200
100 RESTORE 190
110 FOR I = 1 TO 5 READ B: PRINT B: NEXT I
120 GOTO 10
190 DATA 10, 20, 30, 40, 50, 60
200 RESTORE 290
210 FOR I = 1 TO 5 READ B$: PRINT B$: NEXT I
220 GOTO 10
290 DATA "APPLE", "BOY", "CAT", "DOG", "ELEPHANT", "FOX"
```

If a 2 is entered, the first 5 string data values in line 290 are printed; otherwise, the first 5 numeric data values on line 190 are printed. The sixth data items in lines 190 and 290 are not read.

4.5 The GOTO Statement

The GOTO statement is used when it is desired to unconditionally transfer to some line other than the next sequential line in the program. In other words, a GOTO statement causes an immediate jump to a specified line, out of the normal consecutive line number order of execution. The general form of the statement is as follows:

```
GOTO line number
```

The line number to which the program jumps can be either greater or lower than the current line number. It is thus possible to jump forward or backward within a program. For example:

```
10 A=2
20 GOTO 50
30 A=SQR(A+14)
50 PRINT A,A*A
RUN
```

causes the following output:

```
 2       4
```

When the program encounters line 20, control transfers to line 50; line 50 is executed, control then continues to the line following line 50. Line 30 is never executed. Any number of lines can be skipped in either direction.

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

```
110 A=ATN(B2):PRINT A:GOTO 50
```

However, REM and DATA statements can follow a GOTO on the same line because they are non-executable statements.

4.6 Relational Operators

Relational operators allow comparison of two values and are usually used to compare arithmetic expressions or strings in an IF...THEN statement. The relational operators are:

MATHEMATICAL SYMBOL	BASIC SYMBOL	EXAMPLE	MEANING
=	=	A=B	A is equal to B.
<	<	A<B	A is less than B.
$\leq$	<= , =<	A<=B	A is less than or equal to B.

>	>	A>B	A is greater than B.
$\geq$	>= , =>	A>=B	A is greater than or equal to B.
$\neq$	<> , ><	A<>B	A is not equal to B.

The result of the relational operators is -1 for true and 0 for false.

4.6.1 Relational Operators in Strings

When applied to string operands, the relational operators test alphabetic sequence. Comparison is made character by character on the basis of the ASCII codes (See Appendix E) until a difference is found. If, while the comparison is proceeding, the end of one string is reached, the shorter string is considered to be smaller. For example:

```
55 IF A$<B$ THEN 100
```

When line 55 is executed, the first characters of each string (A$ and B$) are compared, then the second characters of each string, and so on until the character in A$ is less than the corresponding character in B$. If this test is true, execution continues at line 100. Essentially, the strings are compared for alphabetic order. Below is a list of the relational operators and their string interpretations.

In any string comparison, leading and trailing blanks are significant (i.e., "ABC" is not equivalent to "ABC ").

OPERATOR	EXAMPLE	MEANING
=	A$=B$	The strings A$ and B$ are alphabetically equal.
<	A$<B$	The string A$ alphabetically precedes B$.
>	A$>B$	The string A$ alphabetically follows B$.
<=	A$<=B$	The string A$ is equivalent to or precedes B$ alphabetically.
>=	A$>=B$	The string A$ is equivalent to or follows B$ alphabetically.
<>	A$<>B$	The strings A$ and B$ are not alphabetically equal.

4.7 Logical Operators

Logical operators are typically used as Boolean operators in relational expressions. For example, consider the following two sequences of statements:

```
100 IF A = B THEN 150
110 IF C < D THEN 150
```

and

```
200 IF A <> 5 THEN 220
210 IF B = 10 THEN 250
220 ...
```

In both cases the sequences can be simplified by using the logical operators AND and OR. The first two statements can be combined into a single statement:

```
100 IF A = B OR C < D THEN 150
```

Similarly, the second sequence of statements is equivalent to:

```
200 IF A = 5 THEN IF B = 10 THEN 250
220 ...
```

This can be further simplified to:

```
200 IF A = 5 AND B = 10 THEN 250
```

Following the rules of Boolean algebra, the unary operator NOT will change true into false and vice versa. For example:

```
100 IF A <> 5 THEN 150
```

is equivalent to:

```
100 IF NOT (A=5) THEN 150
```

More complex expressions can be constructed by using combinations of the AND, OR, and NOT operators.

Logical operators may also be used for bit manipulation and Boolean algebraic functions. The AND, OR, and NOT operators convert their arguments into sixteen bit, signed, two's complement integers in the range -32768 to 32767. After the operations are performed, the result is returned in the same form and range. If the arguments are not in this range, a CF error message will be printed and execution will be terminated. Truth tables for the logical operators appear below. The operations are performed bitwise, that is, corresponding bits of each argument are examined and the result computed one bit at a time. In binary operations, bit 7 is the most significant bit of a byte and bit 0 is the least significant.

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X AND Y
1	1	1
1	0	1
0	1	1
0	0	0

NOT

X	NOT X
1	0
0	1

Some examples will serve to show how the logical operators work:

63 AND 16=16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16
15 AND 14=14	15 = binary 1111 and 14 = binary 1110 , so 15 AND 14 = binary 1110 = 14
-1 AND 8=8	-1=binary 1111111111111111 and 8=binary 1000, so -1 AND 8 = 8
4 OR 2=6	4 = binary 100 and 2 = binary 10 so 4 OR 2 = binary 110 = 6
10 OR 10=10	binary 1010 OR'd with itself is 1010=10
-1 OR -2=-1	-1 = binary 1111111111111111 and -2 = 1111111111111110, so -1 OR -2 = -1
NOT 0=-1	the bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1
NOT X=-(X+1)	the two's complement of any number is the bit complement plus one.

A typical use of logical operations is "masking"--testing a binary number for some predetermined pattern of bits. Such numbers might come from the computer's input ports and would then reflect the condition of some external device.

4.8 The IF...THEN and IF...GOTO Statements

The IF-THEN statement is used to transfer control conditionally from the normal consecutive order of statement numbers, depending upon the truth of some mathematical relation or relations. The basic form of the IF statement is as follows:

```
                 THEN
IF expression          line number
                 GOTO
```

where 'expression' is an arithmetic expression. If the result of the expression is nonzero (true), execution begins at the line number given and proceeds as usual. If the value of the expression is zero (false), the next statement in numerical order will be executed. Usually the statement is of the form:

```
                                        THEN
IF expression  rel. op. expression            line number
                                        GOTO
```

In this case, expressions cannot be mixed; both must be string or both must be numeric. Numeric comparisons are handled as described in 4.6. String comparisons are performed on the ASCII values of the strings as described in 4.6.1 and Appendix E. The rel. op. (relational operator) must be, as described in 4.6, and the line number is the line of the program to which control is conditionally passed.

If the value of the expression is true, control passes to the line number specified. If the value of the expression is false, control passes to the next statement in sequence. For example:

```
30 IF A=B THEN 20
40 PRINT A+B
50 PRINT A^2
```

```
40 IF A<>71 GOTO 20
55 PRINT A
60 D=A+B+*C
```

An alternate form of the IF...THEN statement is as follows:

```
IF expression THEN statement
```

where the statement is any valid DISK BASIC statement. Note that multiple statements can follow the THEN if they are separated by colons (:). With this form of the IF...THEN statement, if the expression evaluates to non-zero (true), the statements following the THEN are executed. Otherwise, control passes to the next numbered line. For example:

```
10 A=10
20 IF A=10 THEN PRINT "TRUE":GOTO 40
30 PRINT "FALSE"
40 END
```

4.9 The FOR and NEXT Statements

FOR and NEXT statements define the beginning and end of a loop. (A loop is a set of instructions which are repeated over and over again, each time being modified in some way until a terminal condition is reached.) The FOR statement is of the form:

```
FOR variable = expression1 TO expression2 STEP expression3
```

where the variable is the index, expression1 is the initial value, expression2 is the terminal value, and expression3 is the incremental value. For example:

```
15 FOR K=2 TO 20 STEP 2
```

causes the program to execute the designated loop as long as K is less than or equal to 20. Each time through the loop, K is incremented by 2, so the loop is is executed a total of 10 times. After executing the loop, when K=20, program control passes to the line following the associated NEXT statement, and the value of K is 22.

The index variable must be unsubscripted, although such loops are commonly used in dealing with subscripted variables. In such a case the control variable is used as the subscript of a previously defined variable. The expressions in the FOR statement can be any acceptable BASIC expression.

The NEXT statement signals the end of the loop which began with the FOR statement. The NEXT statement is of the form:

```
NEXT variable
```

where the variable is the same variable specified in the FOR statement. The variable is actually optional, since any NEXT statement encountered is assumed by the computer to be closing the loop for the appropriate FOR variable. Together the FOR and NEXT statements define the boundaries of a program loop. When execution encounters the NEXT statement, the computer adds the STEP expression value to the variable and checks to see if the variable is still less than or equal to the terminal expression value. When the variable exceeds the terminal expression value, control falls through the loop to the statement following the NEXT statement. Note that the variable is not necessary since when a NEXT statement is encountered it is assumed it is for the appropriate FOR loop variable.

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

The expressions within the FOR statement are evaluated once upon initial entry into the loop. The test for completion of the loop is made after each execution of the loop. (If the test fails initially, the loop is still executed once.)

The index variable can be modified within the loop. When control falls through the loop, the index variable retains the value used to fall through the loop.

The following is a demonstration of a simple FOR-NEXT loop. The loop is executed 10 times; the value of I is 11 when control leaves the loop; and +1 is the assumed STEP value:

```
10 FOR I=1 TO 10
20 PRINT I
30 NEXT I
40 PRINT I
```

The loop itself is defined by lines 10 through 30. The numbers 1 through 10 are printed when the loop is executed. After I=10, control passes to line 40 which causes 11 to be printed. If line 10 had been:

```
10 FOR I = 10 TO 1 STEP -1
```

the value printed by line 40 would have been 0.

The following loop is executed only once since the value of I=44 has been reached and the termination condition is satisfied.

```
10 FOR I = 2 TO 44 STEP 2
20 I = 44
30 NEXT I
```

If the initial value of the variable is greater than the terminal value, the loop is still executed once. The loop set up by the statement:

```
10 FOR I = 20 TO 2 STEP 2
```

will be executed only once although a statement like the following will initialize execution of a loop properly:

```
10 FOR I = 20 TO 2 STEP -2
```

For positive STEP values, the loop is executed until the control variable is greater than its final value. For negative STEP values, the loop continues until the control variable is less than its final value.

FOR loops can be nested but not overlapped. The depth of nesting depends upon the amount of user storage space available; in other words, upon the size of the user program and the amount of RAM available. Nesting is a programming technique in which one or more loops are completely within another loop. The field of one loop (the numbered lines from the FOR statement to the corresponding NEXT statement, inclusive) must not cross the field of another loop. For example:

ACCEPTABLE NESTING TECHNIQUES	UNACCEPTABLE NESTING TECHNIQUES

Two-Level Nesting

Acceptable:

```
10 FOR I1 = 1 TO 10
20 FOR I2 = 1 TO 10
30 NEXT I2
40 FOR I3 = 1 TO 10
50 NEXT I3
60 NEXT I1
```

Unacceptable:

```
10 FOR I1 = 1 TO 10
20 FOR I2 = 1 TO 10
30 NEXT I1
40 NEXT I2
```

Three-Level Nesting

Acceptable:

```
10 FOR I1 = 1 TO 10
20 FOR I2 = 1 TO 10
30 FOR I3 = 1 TO 10
40 NEXT I3
50 FOR I4 = 1 TO 10
60 NEXT I4
70 NEXT I2
80 NEXT I1
```

Unacceptable:

```
10 FOR I1 = 1 TO 10
20 FOR I2 = 1 TO 10
30 FOR I3 = 1 TO 10
40 NEXT I3
50 FOR I4 = 1 TO 10
60 NEXT I4
70 NEXT I1
80 NEXT I2
```

It is possible to exit from a FOR-NEXT loop without the control variable reaching the termination value. A conditional or unconditional transfer can be used to leave a loop. Control can only transfer into a loop which has been left earlier without being completed, ensuring that termination and STEP values are assigned.

Both FOR and NEXT statements can appear anywhere in a multiple statement line. For example:

```
10 FOR I = 1 TO 10 STEP 5: NEXT I: PRINT "I="; I
```

causes:

```
I= 11
```

to be printed when executed.

In the case of nested loops which have the same endpoint, a single NEXT statement of the following form can be used:

```
NEXT variable 1, ... , variable N
```

The first variable in the list must be that of the most recent loop, the second most recent, and so on. For example:

```
10 FOR I=1 TO 10
20 FOR J=1 TO 10
30 ...
100 NEXT J,I
```

5. FUNCTIONS AND SUBROUTINES

5.1 Functions

BASIC provides functions to perform certain standard mathematical operations which are frequently used and time-consuming to program. These functions have three or four letter call names followed by a parenthesized argument. They are pre-defined and may be used anywhere in a program.

Call Name	Function
ABS(x)	Returns the absolute value of x.
ATN(x)	Returns the arctangent of x as an angle in radians in range $\pm\pi/2$), where π = 3.14159.
CALL(x)	Call the user machine language program at decimal location 33282. (8202 HEX) The D,E registers have value of X upon entry and value of Y upon return from machine language routine.
COS(x)	Returns the cosine of x radians.
EXP(x)	Returns the value of e^x where e = 2.71828.
FRE(x)	Returns the number of free bytes not in use.
INT(x)	Returns the greatest integer less than or equal to x.
INP(x)	Returns a byte from input port x. The range for x is 0 to 255.
LOG(x)	Returns the natural logarithm of x.
PEEK(x)	Returns a byte from memory address -32768<x<65535; or if x is negative the memory address is 65536+x.
POS(x)	Returns the value of the current cursor position between 0 and 63.
RND(x)	Returns a random number between 0 and 1.
SGN(x)	Returns a -1, 0, or 1, indicating the sign of x.

SIN(X)	Returns the sine of x radians.
SPC(x)	Causes x spaces to be generated. (Valid only in a PRINT statement).
SQR(x)	Returns the square root of x.
TAB(x)	Causes the cursor to space over to column number x. (Valid in PRINT statement only).
TAN(x)	Returns the tangent of x radians.

The argument x to the functions can be a constant, a variable, an expression, or another function. Square brackets cannot be used as the enclosing characters for the argument x, e.g. SIN[x] is illegal.

Function calls, consisting of the function name followed by a parenthesized argument, can be used as expressions anywhere that expressions are legal.

Values produced by the functions SIN(x), COS(x), ATN(x), SQR(x), EXP(x), and LOG(x) have six significant digits.

5.1.1 The Sine and Cosine Functions; SIN(x) and COS(x)

The SIN and COS functions require an argument angle expressed in radians. If the angle is stated in degrees, conversion to radians may be done using the identity:

radians = degrees * (π /180)

In the following example program, 3.14159 is used as a nominal value for π. P is set equal to this value at line 20. At line 40 the above relationship is used to convert the input value into radians. Note the use of the TAB function to produce a more legible printout.

```
10 REM CONVERT ANGLE (X) TO RADIANS, AND
11 REM FIND SIN AND COS
20 P = 3.14159
25 PRINT "DEGREES",, "RADIANS",, "SINE",, "COSINE"
30 FOR X = 0 TO 90 STEP 15
40 Y = X*(P/180)
60 PRINT X, Y;TAB(32); SIN(Y); TAB(48); COS(Y)
70 NEXT X

RUN
DEGREES          RADIANS          SINE             COSINE
 0                0                0                1
 15               .261799          .258819          .965926
 30               .523598          .5               .866026
 45               .785398          .707106          .707107
 60               1.0472           .866025          .500001
 75               1.309            .965926          .25882
 90               1.5708           1                1.12352E-06
```

5.1.2 The Arctangent and Tangent Functions; ATN(x) and TAN(x)

The arctangent function returns a value in radian measure, in the range $-\pi/2$ to $+\pi/2$ corresponding to the value of a tangent supplied as the argument (x).

In the following program, the input is an angle in degrees. Degrees are then converted to radians at line 50. At line 70 the tangent value, Z, is supplied as the argument to the ATN function to derive the value found on column 4 of the printout under the label ATN(x). Also in line 70 the radian value of the arctangent function is converted back to degrees and printed in the fifth column of the printout as a check against the input value shown in the first column.

```
10 P = 3.14159
15 PRINT
20 PRINT "ANGLE","ANGLE";TAB(20);"TAN(X)";
21 PRINT TAB(32);"ATAN(X)",,"ATAN(X)"
25 PRINT "(DEGS)", "(RADS)",,,"(RADS)",,"(DEGS)"
30 FOR X = 0 TO 45 STEP 15
35 PRINT
40 FOR X = 0 TO 75 STEP 15
50 Y = X*P/180
60 Z = TAN(Y)
70 PRINT X,Y;TAB(20);Z;TAB(32);ATN(Z);TAB(48);ATN(Z)*180/P
80 NEXT X
RUN
ANGLE   ANGLE       TAN(X)      ATAN(X)         ATAN(X)
(DEGS)  (RADS)                  (RADS)          (DEGS)
 0       0          0           0                0
 15      .21799     .267949     .261799          15
 30      .523598    .57735      .523598          30
 45      .785398    .999999     .785398          45
 60      1.0472     1.73205     1.0472           60
 75      1.309      3.73204     1.309            75
```

5.1.3 The Square Root Function; SQR(x)

This function derives the square root of any positive number as shown below:

```
10 INPUT X
20 X = SQR(X)
30 PRINT X
40 GOTO 10
RUN
?16
 4
?1000
 31.6228
?(LINEFEED) (RETURN)
READY
```

If the argument is negative, a CF error will result.

5.1.4 The Exponential and Logarithmic Functions; EXP(x) and LOG(x)

The exponential function raises the number e to the power x. EXP is the inverse of the LOG function. The relationship is:

```
LOG(EXP(X)) = X = EXP(LOG(X))
```

The following program prints the exponential equivalent of an input value.

```
10 INPUT X
20 PRINT EXP(X), LOG(EXP(X)), EXP(LOG(x))
30 GOTO 10
RUN

?87
 6.07601E+37     87       87
?.0033
 1.00331         3.2999E-03      3.3E-03
?1
 2.71828         1        1
```

Logarithms to the base e may easily be converted to any other base using the following formula:

$$\log_a N = \frac{\log_e N}{\log_e a}$$

where a represents the desired base and e = 2.71828. The following program illustrates conversion to the bases 10 and 2.

```
10 PRINT "VALUE","BASE E LOG","BASE 10 LOG","BASE 2 LOG"
20 INPUT X
30 PRINT X,LOG(X);TAB(24);LOG(X)/LOG(10);
40 PRINT TAB(40);LOG(X)/LOG(2)
50 GOTO 20
RUN
VALUE   BASE E LOG      BASE 10 LOG     BASE 2 LOG
?1
 1       0               0               0
?4
 4       1.38629         .60206          2
?10
 10      2.30259         1               3.32193
?1000
 1000    6.90776         3               9.96579
```

An attempt to find the LOG of zero or of a negative number causes a CF error message.

5.1.5 The Absolute Value Function; ABS(x)

The ABS function returns the absolute value of any argument. The absolute value is the argument itself with a positive sign. For example the absolute value of both 3 and -3 is 3. The ABS function may be illustrated as follows:

```
PRINT ABS(12.34),ABS(-23.65)
 12.34   23.65
```

5.1.6 The Greatest Integer Function; INT(x)

The greatest integer function returns the value of the greatest integer not greater than x. For example:

```
PRINT INT(34.67)
 34

PRINT INT(11)
 11
```

The INT of a negative number is a negative number with the same or larger absolute value, i.e., the same or smaller algebraic value. For example:

```
PRINT INT(-23.45)
-24

PRINT INT(-11)
-11
```

The INT function can be used to round numbers to the nearest integer, using INT(X+.5). For example:

```
PRINT INT(34.67+.5)
 35

PRINT INT(-5.1+.5)
-5
```

INT(x) can also be used to round to any given decimal place or integral power of 10, by using the following expression as an argument:

```
(X*10^D+.5)/10^D
```

where D is an integer supplied by the user.

```
10 REM INT FUNCTION EXAMPLE
15 PRINT
20 PRINT "NUMBER TO BE ROUNDED:"
25 INPUT A
40 PRINT "NO. OF DECIMAL PLACES:"
45 INPUT D
60 B = INT(A*10^D +.5)/10^D
70 PRINT "NUMBER ROUNDED = " ;B
80 GOTO 15
RUN

NUMBER TO BE ROUNDED
?55.65842
NO. OF DECIMAL PLACES:
?2
NUMBER ROUNDED =  55.66

NUMBER TO BE ROUNDED
?78.375
NO. OF DECIMAL PLACES:
?-2
NUMBER ROUNDED =  100

NUMBER TO BE ROUNDED
?67.38
NO. OF DECIMAL PLACES:
?-1
NUMBER ROUNDED =  70

NUMBER TO BE ROUNDED
?(LINEFEED) (RETURN)

READY
```

5.1.7 The Random Number Function; RND(x)

The random number function produces a random number, or random number set between 0 and 1. The numbers are reproducible in the same order after the ESC, E sequence if X>0 for later checking of a program. In DISK BASIC the form RND without arguments is not legal. For example:

```
10 PRINT "RANDOM NUMBERS:
30 FOR I = 1 TO 8
40 PRINT RND(I),
50 NEXT I
RUN
RANDOM NUMBERS:
 .100250         .968134         .886657         .636444
 .839019         .306121         .285553         .285534
```

To obtain random digits from 0 to 9, line 40 can be changed to read:

```
40 PRINT INT(10*RND(1)),
```

This time the results will be printed as follows:

```
RANDOM NUMBERS:
 8       9       3       5       6       1       8
 2
```

It is possible to generate random numbers over a given range. If the open range (A,B) is desired, use the expression:

```
(B-A)*RND(1)+A
```

to produce a random number in the range A<n<B.

The following program produces a random number set in the open range (4,6). The extremes, 4 and 6, are never reached.

```
10 REM RANDOM NUMBER SET IN OPEN RANGE 4,6.
20 FOR B = 1 TO 8
30 A = (6-4) * RND(1) + 4
40 PRINT A,
50 NEXT B
RUN
 4.20054         5.92962         5.77325         5.27288
 4.99125         5.02420         4.18825         5.99989
```

Negative arguments, i.e. RND(-123), will start a new random number sequence, while RND(0) will always generate the last random number.

5.1.8 The Sign Function; SGN(x)

The sign function returns the value 1 if x is a positive number, 0 if x is 0 and -1 if x is negative. For example:

```
10 REM SGN FUNCTION EXAMPLE
20 READ A,B,C
25 PRINT "A = "A,"B = "B,"C = "C
30 PRINT "SGN(A) = "SGN(A), "SGN(B) = "SGN(B),
40 PRINT "SGN(C) = "SGN(C)
50 DATA -7.32, .44, 0
RUN
A = -7.32       B =  .44        C =  0
SGN(A) = -1     SGN(B) =  1     SGN(C) =  0
```

5.1.9 The Position Function; POS(x)

The POS function returns the current x coordinate of the cursor's position. It is most often used to determine whether or not a particular program result, either string or numeric, will fit on a given line. By use of the POS(x) function, the correct placement of the answer can be easily determined.

5.2 User Defined Functions

In some programs it may be necessary to execute the same sequence of statements or mathematical formulas in several different places. BASIC allows definition of unique operations or expessions and the calling of these functions in the same way as the predefined standard mathematical functions.

These user defined functions are described by a function name, the first two letters of which are FN followed by any acceptable BASIC variable name. For example:

Legal	Illegal
FNA	FNA$
FNAA	FN2
FNA1	

Each function is defined once and the definition may appear anywhere in the program. The defining or DEF statement is formed as follows:

```
DEF FNA (argument) = expression
```

where A is a variable name. The argument must be a simple variable. The expression may contain the argument variable and any other program variables. For example:

```
10 DEF FNA(S) = S^2
```

causes a later statement:

```
20 R = FNA(4)+1
```

to be evaluated as R = 17. As another example:

```
50 DEF FNB(A) = A+X^2
60 Y= FNB(14)
```

causes the function to be evaluated with the current value of the variable X within the program. The two following programs:

```
10 DEF FNS(A) = A^A          10 DEF FNS(X) = X^X
20 FOR I=1 TO 5              20 FOR I=1 TO 5
30 PRINT I, FNS(I)           30 PRINT I, FNS(I)
40 NEXT I                    40 NEXT I
```

cause the same output:

```
RUN
 1        1
 2        4
 3        27
 4        256
 5        3125
```

User defined functions cannot have several arguments, as shown below:

```
25 DEF FNL(X,Y,Z) = SQR(X^2 + Y^2 + Z^2)
```

Such a statement will cause an error of the type:

```
SN ERROR IN 25
```

When calling a user defined function, the parenthesized argument can be any legal expression. The value of the expression is substituted for the argument variable. For example:

```
10 DEF FNZ(X) = X^2
20 A=2
30 PRINT FNZ(2+A)
```

Line 30 causes the result 16 to be printed.

If the same function name is defined more than once, then the last defintion (the one with the higher line number) will be used. The program below:

```
10 DEF FNX(X) = X^2
20 DEF FNX(X) = X+X
30 A=5
40 PRINT FNX(A)
```

will cause 10 to be printed.

The function variable need not appear in the function expression as shown below:

```
10 DEF FNA (X) = 4+2
20 R=FNA(10)+1
30 PRINT R
RUN
 7
```

5.3 BASIC String Functions

Like the intrinsic mathematical functions described above, BASIC contains various functions for use with character strings. These functions allow the program to concatenate two strings, access part of a string, determine the number of characters in a string, generate a character string corresponding to a given number or vice versa, and perform other useful operations. The various functions available are summarized in the following table.

STRING FUNCTIONS

Call Name	Function
ASC(x$)	Returns the eight bit internal ASCII code (0-255) for the one-character string. If the argument contains more than one character, then the code for the first character in the string is returned. A value of 0 is returned if the argument is a null string (LEN(x$)=0). (See ASCII codes in Appendix E).
CHR$(x)	Generates a one-character string having the ASCII value of x where x is a number greater than or equal to 0 and less than or equal to 255. Only one character can be generated.
FRE(x$)	Returns number of free string bytes. (See CLEAR statement in 3.11)
LEFT$(x$,I)	Returns left-most I characters of string (x$). If I>LEN(x$), then x$ is returned.
LEN(x$)	Returns the number of characters in the the string x$, with non-printing characters and blanks being counted.
MID$(x$,I,J)	J is optional. Without J, returns right-most characters from x$ beginning with the Ith character. If I>LEN(x$), MID$ returns the null string. With 3 arguments, it returns a string of length J of characters from x$ beginning with the Ith character. If J is greater than the number if characters in x$ to the right of I, MID$ returns the rest of the string. Argument ranges: 0<I<=255, 0<=J<=255.

RIGHT$(x$,I)	Returns right-most I characters of string (x$). If I>LEN(x$), then x$ is returned.
STR$(x)	Returns the string which represents the numeric value of x as it would be printed by a PRINT statement.
VAL(x$)	Returns the number represented by the string x$. If the first character of x$ is not +, -, or a digit, then the value 0 is returned.

In the above example, x$ and y$ represent any legal string expressions, and I and J represent any legal arithmetic expressions.

NOTE: Unlike the mathematical functions, character string functions cannot be defined by the user. Similar results can be obtained by the use of subroutines, as described in Section 5.4.

5.4 Subroutines

A subroutine is a section of a program performing some operation required at more than one point in the program. Sometimes a complicated I/O operation for a volume of data, a mathematical evaluation which is too complex for a user defined function, or any number of other processes may be best performed in a subroutine.

More than one subroutine can be used in a single program, in which case they are best placed one after the other in line number sequence before the DATA statements. It is a useful practice to assign distinctive line numbers to subroutines. For example, if the main program uses line numbers up to 199, use 200 and 300 as the first line numbers of two subroutines. When subroutines are included in a program, the program begins execution and continues until it encounters a GOSUB statement of the form:

```
GOSUB line number
```

where the line number following the word GOSUB is that of the first line of the subroutine. Control then transfers to that line of the subroutine. For example:

```
50 GOSUB 200
```

Control is transferred to line 200 in the user program. The first line in the subroutine can be a remark or any other valid BASIC statement.

Having reached the line containing a GOSUB statement, control transfers to the line indicated after GOSUB; the subroutine is processed until BASIC encounters a RETURN statement of the form:

```
RETURN
```

which causes control to return to the statement following the original GOSUB statement. A subroutine must always be exited via a RETURN statement.

Before transferring to the subroutine, BASIC internally records the next sequential statement to be processed after the GOSUB statement; the RETURN statement is a signal to transfer control to this statement. In this way, no matter how many subroutines there are or how many times they are called, BASIC always knows where to transfer control next. The following program demonstrates the use of GOSUB and RETURN.

```
1  REM THIS PROGRAM ILLUSTRATES GOSUB AND RETURN
10 DEF FNA(X) = ABS(INT(X))
20 INPUT A,B,C
30 GOSUB 100
40 A=FNA(A)
50 B=FNA(B)
60 C=FNA(C)
70 PRINT
80 GOSUB 100
90 END
100 REM THIS SUBROUTINE PRINTS OUT THE SOLUTIONS
110 REM OF THE EQUATION: AX^2 + BX + C = 0
120 PRINT "THE EQUATION IS "A "*X^2 + " B"*X + "C
130 D=B*B -4*A*C
140 IF D<>0 THEN 200
150 PRINT"ONLY ONE SOLUTION...X "; -B/(2*A)
160 RETURN
170 IF D<0 THEN 200
180 PRINT "TWO SOLUTIONS...X=";
185 PRINT (-B+SQR(D))/(2*A); ") AND ("; (-B-SQR(D))/(2*A)
190 RETURN
200 PRINT "IMAGINARY SOLUTIONS...X =(";
205 PRINT -B/(2*A) "," SQR(-D)/(2*A) ") AND (";
207 PRINT -B/(2*A) ","; -SQR(-D)/(2*A) ")"
210 RETURN
900 END
```

Subroutines can be nested; that is, one subroutine can call another subroutine. If the execution of a subroutine encounters a RETURN statement, it returns control to the statement following the GOSUB which called that subroutine. Therefore, a subroutine can call another subroutine, even itself. Subroutines can be entered at any point and can have more than one RETURN statement. It is possible to transfer to the beginning or any part of a subroutine; multiple entry points and RETURN's make a subroutine more versatile.

5.5 The ON GOTO and ON GOSUB Statements

The ON...GOTO statement provides another type of conditional branching. Its form is as follows:

```
ON  expression  GOTO  line number  list
```

After the value of the expression is truncated to an integer in the range 0-255, say I, the statement causes BASIC to branch to the line whose number is Ith in the list. If I=0 or is greater than the number of lines in the list, execution will continue at the next line after the ON...GOTO statement. If I is less than 0 or greater than 255, a CF error will result. For example, the following sequence of IF statements can be replaced by a single ON...GOTO statement. Thus;

```
100 IF X=1 THEN 1000
110 IF X=2 THEN 2000
120 IF X=3 THEN 3000
130 IF X=4 THEN 4000
140 IF X=6 THEN 6000
150 Y=10
```

can be replaced by:

```
100 ON X GOTO 1000, 2000, 3000, 4000, 150, 6000
150 Y=10
```

Note that there was no IF statement for X=5, so in the ON...GOTO statement the corresponding line number is 150, which is the next line.

Subroutines may be called conditionally by use of the ON...GOSUB statement. Its form is as follows:

```
ON  expression  GOSUB  line number  list
```

The execution is the same as ON...GOTO except that the line numbers are those of the first lines of subroutines. Execution continues at the next statement after the ON...GOSUB upon return from one of the subroutines.

Note that ON...GOTO and ON...GOSUB statements do not have to be the last executable statements on a line.

6. ARRAYS

6.1 Introduction to Arrays

Arrays or subscripted variables are most frequently used for storing lists of information in a program using a single name to refer to the list as a whole and using subscripts to refer to individual items. For example, consider the following list of 12 numbers corresponding to the number of days in each month in a non-leap year:

31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31

The notion of subscripts follows naturally. For instance, the 5th item in the list corresponds to the number of days in May. Using an array (list) of size 12, named M, to refer to all the entries in the list as a whole, the fifth item of M can be simply denoted as M(5). Similarly, the number of days in February is denoted by M(2). If the number of days in the Ith month is desired, then M(I) contains that value.

In the following example, the data values are read into an array which is dimensioned to size 12 in line 10. (See Section 6.4)

```
10 DIM M(12)
20 FOR I=1 TO 12: READ M(I): NEXT I
30 DATA 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
35 REM PRINT THE NUMBER OF THE MONTH AND DAYS IN EACH MONTH
36 REM ADD UP THE NUMBER OF DAYS IN THE MONTHS
40 D=0
50 FOR I=1 TO 12
60 PRINT I, M(I)
70 D=D+M(I)
80 NEXT I
90 PRINT "TOTAL DAYS =", D
```

The resulting output from this program is:

```
RUN
 1       31
 2       28
 3       31
 4       30
 5       31
 6       30
 7       31
 8       31
 9       30
 10      31
 11      30
 12      31
TOTAL DAYS =    365
```

If the above program were expanded past line 90 the values in M would be accessible at any point during the execution of the program unless they were changed by an assignment or input statement.

6.2 Subscripted Variables

The name of a subscripted variable is any acceptable BASIC variable name followed by one or more integer expressions in parentheses within the range 0 - 32767. Subscripted variable names follow the same naming conventions as simple variables with the first 2 characters being significant. For example, a list might be described as A(I), where I goes from 0 to 5 as shown below:

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

This allows reference to each of the six elements in the list, and can be considered a one dimensional algebraic matrix as follows:

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

A two-dimensional matrix B (I,J) can be defined in a similar manner:

```
B(0,0),B(0,1),B(0,2),...,B(0,J),...B(I,J)
```

and graphically illustrated as follows:

```
B(0,0)  B(0,1)  ...  B(0,J)
B(1,0)  B(1,1)  ...  B(1,J)
   .       .           .
   .       .           .
   .       .           .
B(I,0)  B(I,1)  ...  B(I,J)
```

Higher dimensional arrays can also be formed. The upper limit is determined by the size of the input buffer giving a practical limit of 40.

Subscripts used with subscripted variables throughout a program can be explicitly stated or they can be any legal expression. If the value of the expression is non-integer, the value is truncated so that the subscript is an integer.

It is possible to use the same variable name as both a subscripted and unsubscripted variable. Both A and A(I) are valid variables and can be used in the same program. The variable A has no relationship to any element of the matrix A(I). Subscripted arrays of character strings may also be defined, and their variable names are distinct. A$(I) bears no relation to A(I) or A.

A dimension (DIM) statement is used with subscripted variables to define the maximum number of elements in a matrix.

If a subscripted variable is used without appearing in a DIM statement, it is assumed to be dimensioned to length 10 in each dimension (that is, having eleven elements in each dimension, 0 through 10). However, all matrices should be correctly dimensioned in a program.

6.3 Subscripted String Variables

Any list or matrix variable name followed by the $ character denotes the string form of that variable. For example:

```
V$(n)           M2$(n)
C$(m,n)         G1$(m,n)
```

where m and n indicate the position of the matrix element within the whole.

The same name can be used as a numeric variable and as a string variable in the same program with no restriction. Simple variables and dimensioned variables can also have the same name. For example:

```
A               A(n)
A$              A$(m,n)
```

can all be used in the same program; however, A(n,m) could not be used, because it redefines the size of A(n).

String lists and matrices are defined with the DIM statement as are numerical lists and matrices.

6.4 The DIM Statement

The DIM statement is used to define the maximum number of elements in a matrix. The DIM statement is of the form:

```
DIM variable(n), variable(n,m), variable$(n), variable$(n,m)
```

where variables specified are indicated with their maximum subscript value(s). For example:

```
10 DIM X(5),Y(4,2), A(10,10)
12 DIM A4(100), A$(25)
```

Arrays can be dynamically dimensioned by using numeric expressions instead of integer constants to define the size of an array. Any number of matrices can be defined in a single DIM statement as long as they are separated by commas.

The first element of every matrix is automatically assumed to have a subscript of zero. Dimensioning A(6,10) sets up room for a matrix with 7 rows and 11 columns. This zero element is illustrated in the following program:

```
10 REM MATRIX CHECK PROGRAM
20 DIM A(6,4)
30 FOR I=0 TO 6
40 A(I,0) = I
50 FOR J=0 TO 4
60 A(0,J) =J
70 PRINT A(I,J);
80 NEXT J:PRINT:NEXT I
90 END
RUN
 0 1 2 3 4
 1 0 0 0 0
 2 0 0 0 0
 3 0 0 0 0
 4 0 0 0 0
 5 0 0 0 0
 6 0 0 0 0
```

Notice that a variable has a value of zero until it is assigned another value.

Whenever an array is dimensioned (m,n), the matrix is allocated with (m+1,n+1) elements. Memory space can be conserved by using the 0th element of the matrix. For example, DIM A(5,9) dimensions a 6 * 10 martrix which would then be referenced beginning with the A(0,0) element.

The size and number of matrices which can be defined depend upon the amount of storage space available.

A DIM statement can be placed anywhere in a multiple statement line and can appear anywhere in the program. A matrix can only be dimensioned oncé. DIM statements must appear prior to the first reference to an array. DIM statements are generally among the first statements of a program to allow them to be easily found if any alterations are later required.

All arrays specified in DIM statements are allocated space when the DIM statement is executed. All other arrays are declared at the first reference executed.

7. FURTHER SOPHISTICATION

7.1 Formatting the Printout

Often, the purpose of a program will require that results be printed out in a particular format, rather than simply in a list or line at the end of a program run. BASIC provides certain facilities for use in formatting the printout, so that the desired result can be achieved.

When a comma separates a text string from another PRINT list item, the item is printed at the beginning of the next available print zone. Semicolons separating text strings from other items are ignored. The screen is divided into 8 print zones of 8 characters each. A comma or semicolon appearing as the last item of a PRINT list always suppresses the carriage return/line feed operation. BASIC does an automatic carriage return/line feed if a string is printed past column 64. Examples of the use of comma include:

```
10 A=3
20 B=2
30 PRINT A,B,A+B,A*B,A-B,B-A
```

When the preceding lines are executed, the computer will print:

```
 3       2       5       6       1      -1
```

Notice that each character is eight spaces from the next character. Two commas together in a PRINT statement cause a print zone to be skipped, as in:

```
10 A=1
20 B=2
30 PRINT A,B,,A+B
RUN
 1       2               3

READY
```

If the last item in a PRINT statement is followed by a comma, no carriage return/linefeed is output, and the next value to be printed (by a later PRINT statement) appears in the next available print zone. For example:

```
10 A=1:B=2:C=3
20 PRINT A, :PRINT B: PRINT C
RUN
 1       2
 3
READY
```

If a tighter packing of printed values is desired, the semicolon can be used in place of the comma. A semicolon causes no spaces to be output other than the leading space automatically output with each non-negative number. A comma causes the cursor to move at least one space to the next print zone or perform a carriage return/line feed if the string prints past column 64. The following example shows the effects of the semicolon and comma.

```
10 A=1:B=2:C=3
20 PRINT A;B;C;
30 PRINT A+1;B+1;C+1
40 PRINT A,B,C
RUN
 1 2 3 2 3 4
 1       2       3
READY
```

The following example demonstrates the use of the formatting characters , and ; with text strings:

```
120 PRINT "STUDENT"X; " GRADE ="G;" AVG. ="A;
130 PRINT " NO. IN CLASS ="N
```

Assuming that calculations had been done prior to these lines, the following would result:

```
STUDENT 119050 GRADE = 87 AVG. = 85.44 NO. IN CLASS = 26
```

7.1.1 The Tabulator Function; TAB(x)

The TAB function is used in a PRINT statement to write spaces to the specified colomn on the output device. The columns on the screen are numbered 1 to 64. The form of the command is:

```
PRINT TAB(x)
```

where (x) is the column number in the range 0 - 255. (If x exceeds 64, however, every other consecutive line is tabbed until the number of specified spaces are printed. If (x) is greater than 255 or negative, an error message is printed as follows:

```
CF ERROR
READY
```

If (x) is non-integer, only the integer portion of the number is used. If the column number (x) specified is less than or equal to the current column number, the TAB function has no effect.

7.1.2 The Space Function; SPC(x)

The SPC function can be used in much the same fashion as TAB in PRINT statements. This function prints the number of spaces indicated by (x) which must be in the range 0-255; otherwise a CF error results.

Note that if either a TAB(x) or SPC(x) is the last item in a print list the carrige return/line feed is suppressed.

7.2 Immediate Mode and Debugging

Immediate mode operation is especially useful for program debugging (error removal), and performing simple calculations in situations which do not occur with sufficient frequency or with sufficient complication to justify writing a program.

In order to facilitate debugging a program, END statements can be liberally placed throughout the program. Each END statement causes the program to halt, at which time the various data values can be examined and perhaps changed in immediate mode. The command:

```
GOTO xxxxx
```

is used to continue program execution (where xxxxx is the number of the next program line to be executed). GOSUB and IF commands can also be used. The values assigned to the variables when the RUN command is executed remain intact until a CLEAR statement or another RUN command is executed.

When using immediate mode, nearly all of the standard statements can be used to generate or print results.

If LINEFEED is used to halt program execution, the GOTO xxxx or CONT command can be used to continue execution. Since CTRL/J or LINEFEED does print the number of the line where execution stopped, it is easy to know where to resume the program. Note that if a BASIC program statement is entered or altered, it is not possible to continue execution.

7.2.1 Restrictions on Immediate Mode

The INPUT and DEF statements cannot be used in immediate mode and such use results in the following error message:

```
ID ERROR
READY
```

Certain other commands, while not illegal, make no logical sense when used in immediate mode. Commands in this category are DIM and DATA.

Although the standard mathematical functions are permissible, user functions are not defined until the program is executed, and therefore any references to user defined functions in immediate mode cause an error unless the program containing the definition was previously executed.

Thus, the following dialogue might result if a function were defined in a user program and then referenced in immediate mode.

```
10 DEF FNA(X) = X^2 + 2*X:REM SAVED STATEMENT
PRINT FNA(1):REM IMMEDIATE MODE

UF ERROR
READY
```

but if the sequence of statements were:

```
10 DEF FNA(X) = X^2+2*X:REM SAVED STATEMENT
RUN

READY

PRINT FNA(1)
 3

READY
```

the immediate mode statement would be executed.

7.3 Machine Level Interfaces with DISK BASIC

DISK BASIC has several features that allow the user access to the machine level input/output of the microprocessor. By using the WAIT and OUT statements and the INP function, various input/output operations can be performed. Other machine dependent features allow access to the memory and assembly language subprograms. (See Appendices D.1 and D.2 for Key Memory Locations and Port Assignments.)

7.3.1 The WAIT Statement

The status of memory ports can be monitored by the WAIT statement which has the following forms:

```
WAIT I,J
WAIT I,J,K
```

where I is the number of the port being monitored, and J and K are integer expressions. The port status is exclusive OR'ed with K if present and the result is AND'ed with J. Execution is suspended until a non-zero value results. In other words, J picks the bits of port I to be tested and execution resumes at the next statement after the WAIT. If K is omitted, it is assumed to be zero. I, J, and K must be in the range 0 to 255; otherwise, a CF error results.

7.3.2 The OUT Statement

The form of the OUT statement is as follows:

```
OUT I,J
```

where I and J are integer expressions in the range 0 to 255. OUT sends the 8 bit quantity (byte) signified by J to output port I.

WARNING: If bytes are output to ports on the SMC 5027 CRT chip, serious damage can result to the COMPUCOLOR II. (See Appendix D.2)

7.3.3 The Input Function; INP(x)

The INP function is the counterpart of the OUT statement. Its form is as follows:

```
X = INP(I)
```

INP reads a byte (8 bit quantity) from port I where I is an integer expression in the range 0 to 255.

7.3.4 The Peek Function; PEEK(x)

The PEEK Function is called as follows:

```
J = PEEK(I)
```

where J is the integer value returned in the range 0-255 that is to be stored in the memory location specified by the integer expression I. The range of I is -32768 to 65535. If I is negative, then the address is 65536+I; and if I is positive, the address is I.

7.3.5 The POKE Statement

The form of the POKE statement is as follows:

```
POKE I,J
```

where J is an integer expression in the range 0 to 255 that is to be stored in the memory location specified by the integer expression I. The range of I is -32768 to 65535. If I is negative, then the address is 65536+I; and if I is positive, the address is I.

7.3.6 The User Call Function; CALL(x)

The CALL function is used for interfacing with 8080 machine language subroutines. The function can be used in the same manner as the other mathematical functions. The form is as follows:

```
Y = CALL(x)
```

where the assignment x must be in the range -32768 to 65535. The value Y returned is in the range -32768 to 32767.

The CALL function converts the argument into a 2 byte integer and stores the result in the 8080's D and E registers (D contains the high byte, E the low byte.) The BASIC interpreter then executes an 8080 CALL instruction to location 33282 (8202 HEX), which, unless modifed by the user, contains a jump to the CF ERROR message routine. The user must modify the locations 33282 through 33284 so that they contain a JMP to the desired machine language routine. Upon return, the 2 byte integer in the D,E registers is converted back into floating point format. The stack level must be preserved at the same point at which the user entered the CALL, and the H and L registers must be preserved. All other 8080 registers can be modified.

For example, consider the following assembly language subroutine which negates the contents of the D and E registers.

```
         ORG     08202H  ;33282
         JMP     NEGATE

         ORG     09FF0H  ;40944
NEGATE:  MOV     A,D     ;COMPLEMENT
         CMA             ;HIGH
         MOV     D,A     ;BYTE
         MOV     A,E     ;COMPLEMENT
         CMA             ;LOW
         MOV     E,A     ;BYTE
         INX     D       ;INCREMENT AND  FORM 2'S COMPLEMENT
         RET             ;RETURN - HL UNCHANGED
```

This subroutine could be assembled using the COMPUCOLOR II Assembler or "hand" assembled and entered using the POKE statement in BASIC.

To enter this subroutine in BASIC, the user must first hit CPU RESET then re-enter BASIC by using the ESCAPE W sequence. The number 8176 must be entered in response to the MAXIMUM RAM AVAILABLE prompt. This leaves 16 bytes free for the machine language subroutine. The following program loads the machine language subroutine and demonstrates the CALL function.

```
5  REM CHANGE JUMP ADDRESS AT 8203-4 HEX, 8202H CONTAINS JUMP
10 POKE 33283, 240 : POKE 33284, 159
15 REM PROGRAM BYTES AT 9FF0 HEX
20 DATA 122, 47, 87, 123, 47, 95, 19, 201
30 FOR AD = 40944 TO 40951
40 READ VL: POKE AD, VL
50 NEXT AD
100 INPUT "ENTER X ";X : Y=CALL(X)
110 PRINT "-X = ";Y : GOTO 100
```

7.4 String Space Allocation

Understanding how the string space is used is important in deciding how much string space is necessary for the execution of a program. First, all strings entered in immediate mode or by the INPUT statement (see Section 4.1) are allocated in the string space because the input line buffer can be modifed by subsequent inputs.

String functions and the string concatenation operator "+" always return their results in the string space. Assigning a string a constant value in a program through a READ or assignment statement does not use any string space since the string value is part of the program itself. In general, copying is done when a string value is in the input line buffer, or it is in the string space and there is an active reference to it by a string variable. Thus, A$ = B$ will cause copying if B$ has its string data in the string space. The assignment A$ = STR$(105) (see Section 5.3 for STR$) will use four bytes of string space to store the new four character string, " 105", created by the STR$ function, but the assignment itself does not cause copying since the only reference to the new string was created as a temporary reference by the formula evaluator. The temporary references disappear when the assignment is done. The copying is done in this manner because the string garbage collection does not allow two references to the same area in the string space.

8. DISK FEATURES

8.1 Loading and Saving Programs

Programs and data can be loaded and saved on the COMPUCOLOR II so that they can be stored and used, edited, or updated in the future. The general forms of the LOAD and SAVE statements are:

```
LOAD  string expression
SAVE  string expression
```

where the string can be a string variable such as A$ or a quoted literal string such as "NAME". There are three FILE types that can be loaded and saved. They are BASIC source (BAS), numeric ARRAYS (ARY), and memory DATA (DAT). If no file type is specified, then the default type is BAS. The BAS file type can be in the form as shown below. Each of the following examples will save the same BASIC source.

```
SAVE "TEST" :REM SAVES BASIC SOURCE WITH NAME TEST ON DISK
SAVE "TEST.BAS"
SAVE "TEST.BAS;1"
SAVE A$: REM WHERE A$ IS A STRING VARIABLE
SAVE "CD1:" + A$: REM WHERE "CD1:" SPECIFIES OPTIONAL DISK
```

Each of the following examples will cause a BASIC source program to be loaded.

```
LOAD "TEST:REM LOADS A BASIC SOURCE PROGRAM BY NAME OF TEST
LOAD "TEST.BAS"
LOAD "TEST.BAS;1"
LOAD "CD1:" + A$: REM WHERE "CD1:" SPECIFIES THE SECOND DISK
LOAD A$:REM WHERE A$ IS A STRING VARIABLE
```

The ARY file type can be in the same form as BAS except that ARY must be in the string after the file name. Also the file name must be a dimensioned or previously used array by the same first two letters of the file name. If a one letter variable name is used, then the file name must be that letter only.

```
10 DIM ST (100,10),T(3),TT(11,15,38)
20 SAVE "STEST.ARY"
30 SAVE "T.ARY;1"
40 END
```

The above program will save the numbered arrays ST and T. The following program will cause a (100,10) array to be loaded even though it was originally set at 1200, since 1200 > 101 * 11.

```
10 DIM ST (1200)
20 LOAD "STEST.ARY": REM DIM ST (100,10)
30 END
```

The DAT file type can be in the same form as ARY. It will look at the two-byte integer stored in locations 32940 and 32941 (32940 low byte and 32941 high byte) as a pointer to memory. It adds 1 to this pointer and takes the next two bytes in memory as the number of bytes to be loaded into memory or saved on disk. The locations 32940 and 32941 specify the end of BASIC memory space, so all memory above that location can be used to save data via BASIC using the POKE command. Also note that only one DAT file may be read in at any one time without changing the pointers at 32940 and 32941.

Note that it is recommended that programs use the random file capability of DISK BASIC instead of loading and saving DAT files.

8.1.1 Program Chaining

A series of different programs can be executed as a single program by using a technique commonly known as program chaining. In DISK BASIC, two types of program chaining are possible. The first and easiest method uses the LOAD statement in combination with the RUN command as follows:

```
LOAD"PROGRM":RUN
```

Executing this statement in either a program or immediate mode causes the specified BASIC program to be loaded and executed. The RUN command clears all the variables from the previous program. A line number can optionally be specified on the RUN command.

The second method used the LOAD statement in combination with the GOTO statement as follows:

```
LOAD"PROGRM":GOTO line number
```

Executing this statement in a program causes the specified program to be loaded and executed starting at the specified line number in the GOTO command. This method does not clear the variables from the previous program; however, two restrictions must be satisfied to ensure proper exection of the program. First, the program with the largest source in the chain must be loaded and executed first. Second, string variables whose data values where part of the program source will contain incorrect references when subsequent program is loaded because the program source will not be the same as the previous program. If these restrictions are satisfied, then the series of programs should execute properly. Clearly, this second method of program chaining is the least desirable because of the possible difficulties. See Section 7.4 for a description of how strings are allocated before using this method.

8.1.2 MENU Programs

With the COMPUCOLOR II it is possible to create a program that is loaded and executed by pressing a single key. The AUTO key automatically loads and executes a BASIC program called MENU.BAS from the default device which is the internal COMPUCOLOR II disk drive unless the default device has been changed. (See Chapter 10 for further details on device and file specifications.)

The MENU program can be used to run and control a large application system composed of several programs such as a payroll system. In this case the MENU program asks which function is to be performed next and directs the execution to the proper program or section of BASIC code. Similarly, the MENU program can control a number of unrelated applications by displaying a "menu" of applications accessible on the diskette. This technique is used on many of the COMPUCOLOR II diskette albums. Thus, by depressing the AUTO key, the MENU program is loaded and executed displaying a "menu" of programs. The user simply selects a program by number or name, and then the MENU program chains to the desired program. When the selected program is finished it can chain back to the MENU program.

For example, consider the following three programs:

MENU.BAS

```
10 PRINT "MENU PROGRAM"
20 PRINT
30 PRINT"1 - PRINT TABLE OF POWERS"
40 PRINT"2 - PRINT TABLE OF SINE FUNCTIONS"
50 PRINT
60 INPUT "ENTER NUMBER OF DESIRED PROGRAM"; N
70 N = INT(N)
80 IF N<1 OR N>2 THEN 60
90 ON N GOTO 100,200
100 LOAD "POWERS":RUN : REM EXECUTE POWERS
200 LOAD "SINE": RUN : REM EXECUTE TRIG
999 END
```

POWERS.BAS

```
10 PRINT "N","N^2","N^3"
30 FOR N = 1 TO 10
40 PRINT N,N^2,N^3
50 NEXT N
60 PRINT
100 LOAD "MENU":RUN:REM RETURN TO MENU
```

SINE.BAS

```
10 PI = 3.14159265: REM BASIC ROUNDS TO APPROX. 7 PLACES
20 PRINT "DEG","SINE"
30 PRINT
40 FOR DEG = 0 TO 360 STEP 15
50 RAD = DEG * PI/180
60 PRINT DEG,SIN(RAD)
70 NEXT DEG
80 PRINT
100 LOAD "MENU":RUN :REM RETURN TO MENU
```

To try this example the user must first enter and save each of the three programs, being careful to remember to reinitialize BASIC before entering a new program after saving the old one. After saving all three programs the AUTO key must be struck. This causes MENU to be loaded and executed. MENU then asks for either 1 or 2 and executes the selected program. Note that these programs return back to the MENU program after they have performed their specified function. This makes the MENU program an effective tool for controlling and demonstrating a system or diskette of programs.

8.2 Using the File Control System Through BASIC

The PRINT STRING command preceded by PLOT 27 and PLOT 4 or PLOT 68 (ESC,D for FCS DISK) will enable the user to exercise all of the FCS disk commands through BASIC. Therefore, every command available to the File Control System is also available to BASIC, by letting the string become the FCS command. The following examples show how to retrieve a disk directory through BASIC. (For a description of the PLOT statement, see Section 9.1)

```
10 PLOT 27,4
20 PRINT "DIR"
40 END
```

or

```
10 PLOT 27:PRINT"DDIR"
```

or

```
10 PLOT 27:PLOT 68:PRINT A$:REM WHERE A$ IS A
20 REM STRING VARIABLE EQUAL TO DIR.
```

If the directory of the disk were as follows:

```
TEST.ARY;01
TEST.ARY;02
```

then the BASIC program below would delete version 1 of the TEST.ARY file, rename version 2 to version 1, update the array, and save it as version 2 so it can be used again.

```
5 DIM TEST(1000)
10 LOAD "TEST.ARY;2"
20 PLOT 27:PLOT4:REM SELECT FCS MODE
30 PRINT "DELETE TEST.ARY;1"
50 PRINT "RENAME TEST.ARY;2 TO TEST.ARY1"
60 PLOT 27:PLOT 27:REM SELECT VISIBLE CURSOR MODE
80 :REM UPDATE TEST ARRAY
90 SAVE "TEST.ARY"
```

All string functions that are available to BASIC can be used in the PRINT statement containing the FCS command.

To escape from the File Control System and return to one of the other CRT modes, an escape sequence must be given; such as ESC,ESC for visible CRT cursor mode. The FCS responds only to printing ASCII characters and the following control codes:

11	ERASE LINE
13	CARRIAGE RETURN
26	CURSOR LEFT
27	ESCAPE

All other control codes will cause an FCS error if they appear in a string. A complete description of the FCS commands appears in Chapter 10 and Appendix B.1.

8.2.1 Loading and Saving Displays in BASIC

The FCS interface with BASIC makes it very easy to load and save screen displays. To save displays generated by a BASIC program, the COMPUCOLOR II should first be in page mode (see Section 9.4); otherwise, if the screen has scrolled at all then the saved display will be wrapped around. After the display has been generated, the user simply includes a sequence of statements similar to that shown below.

```
900 PLOT 27,4 : REM ENTER FCS
910 PRINT "SAVE SCREEN.DSP 7000 1000"
920 PLOT 27,27 : REM RETURN FORM FCS
```

Line 910 saves a copy of screen refresh memory which is located from 7000 HEX to 7FFF HEX in a file called SCREEN.DSP. The display can now be loaded at any time by executing the following sequence of statements.

```
1000 PLOT 12 : REM UNROLL SCREEN MEMORY BY ERASE PAGE
1010 PLOT 27,4 : REM ENTER FCS
1020 PRINT"LOAD SCREEN.DSP" : REM LOAD DISPLAY
1030 PLOT 27,27 : REM RETURN FROM FCS
```

These two sequences can be tailored to fit any needs by simply changing the line numbers and name of the file containing the display.

Displays that are generated in CRT mode can also be saved using BASIC. Before creating a display, the user should enter the following one line BASIC program.

```
0 PLOT 27,4: PRINT "SAVE SCREEN.DSP 7000 1000" : END
```

When this line is executed it will save the current display. After the display is finished, the cursor should be moved to a section of the display on the left-hand side that is composed of a few short lines of blanks. Set the background color and foreground colors to the background color of the blanks (see Secton 9.2) and then re-enter BASIC by typing ESC E. A READY message will be returned but it will be invisible because the foreground and background colors are the same. Next type RUN. The RUN command executes the one line BASIC program that saves the display.

8.3 Introduction to Random Files

COMPUCOLOR II DISK BASIC has three statements which implement a powerful random access file capability. The FILE statement performs various functions including creating, opening and closing random files. The GET and PUT statements read, write, and update records in a random file.

Random files are organized into physical blocks containing a fixed number of fixed length records. If a physical block is not a multiple of 128, then the excess length up to the next multiple of 128 is not used. The blocking factor and record size of a file can be changed to allow different types of access. For example, a 100 record file of 80 byte records with a blocking factor of 3 will use only the first 240 bytes of 256 available in 2 disk sectors. The last 16 bytes are unused. Logical records do not cross physical block boundaries. Thus, for the 100 record file 2*34=68 sectors are needed. In this case a 102 record file could have been allocated in the same amount of disk space.

There can be up to 127 random files open simultaneously subject to memory limitations. Memory space for files is allocated dynamically from the user's workspace. Each file can contain from 1 to 32767 records and the record size range is 1 to 32767 bytes. The record size must be small enough to fit into the user's workspace giving a practical maximum of 30000 bytes. The default filename type for random files is .RND.

8.4 The FILE Statement

The basic form of the FILE statement is:

```
FILE "string expression", extra information
```

The FILE statement is a versatile statement that has the ability to perform a number of functions. The first character of the string expression determines what the FILE statement will do. The following sections describe the FILE statement's uses and functions.

8.4.1 Random File Creation

The Random File Creation statement is of the form:

```
FILE "N", filename, records, record size, blocking factor
```

where 'filename' is a string expression containing a valid FCS filename; 'records' is the number of logical records (1-32767); 'record size' is the size in bytes of logical records (1-32767); and 'blocking factor' is the number of logical records per physical block (1-255).

The specified file must not exist. If no version number is specified, then FCS will choose the next larger version number. The user is responsible for choosing proper values of the parameters. Any of the file specifications can be overridden when the file is opened with the FILE "R" statement. For example:

```
FILE "N", "CHECKS", 200, 32, 8
```

creates a file containing 200 32-byte records with 8 records per block.

8.4.2 Random File Open

The form of the Random File Open statement is:

FILE "R",file,name,buffers<;records,rec size,blocking factor>

where 'file' is the logical number of the file (1-127), 'name' is a string expression containing a valid FCS filename, and 'buffers' is the number of buffers in memory (1-255).

The items between the angle brackets are optional and redefine the file size. The elements are: 'records', which is the number of logical records (1-32767); 'rec size', which is the size in bytes of logical records (1-32767); and 'blocking factor', which is the number of logical records per physical block (1-255).

The specified file must already exist. It is possible to open any type of file, but they are best created with the FILE "N" statement. Files not created in BASIC can be accessed by overriding the number of records, the record size, and the blocking factor, but the directory will not contain valid information about the number of records, record size, or blocking factor. For example:

FILE "R",1,"CHECKS", 2

opens the file "CHECKS.RND" and allocates enough buffer space for 2 physical blocks or 16 records.

8.4.3 Random File Close

The Random File Close statement is of the form:

FILE "C", file 1 <,...,file N>

where 'file' is the number of the file to be closed. The items between the angle brackets are optional, and merely describe the format for closing more than one file at a time.

Each file that has been opened must be closed to ensure that the buffers in memory are written to the disk if they have been modified. Closing a file frees up its buffer space in memory. For example:

FILE "C", 1

closes file 1.

8.4.4 Dump File Buffers

The form of the Dump File Buffers statement is:

FILE "D", file 1 <,...,file N>

where 'file' is the number of the file (1-127); and the optional items between the angle brackets are other files that can be included in the same statement.

This statement writes any modified buffers to the disk for the specified files. It can be used to ensure that modifications to a file are recorded immediately. It is similar to FILE "C" except that the buffer space is not freed up and the file remains open. For example:

```
FILE "D",4,6
```

writes any modified buffers back to the disk for files 4 and 6.

8.4.5 File Attributes

The form of the File Attributes statement is:

```
FILE "A",file,cur record <,records,rec size,blocking factor>
```

where 'file' is the number of the file (1-127); and 'cur record' is the variable that is assigned the most recently accessed record number. The items between the angle brackets are optional and include 'records', which is the variable that is assigned the number of records in the file; 'rec size', which is the variable that is assigned the record size in bytes; and 'blocking factor', which is the number of logical records per physical block (1-255).

This statement is used when the file size and other attributes of a file are unknown. For example, the attributes of file 1 may be determined as follows:

```
FILE "A", 1, CR, NR, RS, BF
```

8.4.6 File Error Trapping

The form of the File Error Trapping statement is:

```
FILE "T" <,line number>
```

where the optional line number is a line number in the range 0 to 65529.If the file "T" statement is executed with the line number specified, then when a disk error occurs it will be trapped and execution will continue at the specified line number. All information about nested GOSUB's and FOR-NEXT loops will be lost. In most cases this will not be a problem. In the other cases, assuming good programming practices, the disk error will probably be a hardware failure which requires some type of special recovery procedure. If the line number is not specified, then the error trapping facility will be disabled. For example:

```
FILE "T",32000
```

causes the program to go to line 32000 whenever a disk error occurs.

8.4.7 File Error Determination

The form of the File Error Determination statement is:

FILE "E", file, error, line number

where 'file' is the file number at the time of the error (this number may be incorrect for bad file name errors and errors within the FILE "N" statement); 'error' is the disk error number (for explanations see Appendix A.6); and 'line number' is the line number in which the error occurred.

This statement lets the user determine what type of disk error occurred. It is used in conjunction with the FILE "T" statement. For example:

FILE "E", FL, ER, LN

returns the file, error, and line number of the current random file error.

8.5 The GET Statement

The GET statement is of the form:

GET file <,record <,first>> ; variable list

where 'file' is the logical file number (1-127); and the 'variable list' contains one or more of the following entries:

numeric variable - reads 4 bytes into the numeric variable;
string variable [byte count] - reads the specified number of bytes into the string variable. The byte count range is 1 to 255.

The items between the angle brackets are optional and include 'record', which is the record number to be read (if 0 or omitted, then the record number is 1 greater than that used for the last access to the file); and 'first', which is the first byte of the record to be read (1-record size). If no value is given for 'first', then first defaults to 1.

The GET statement allows a file to be randomly accessed. By using the first field, different parts of the record can be immediately accessed. For example:

GET 1,R;ACCOUNT,AMOUNT,DATE,PAYEE$[20]

will read ACCOUNT, AMOUNT, and DATE as numeric entries, and PAYEE as a 20 byte string.

8.6 The PUT Statement

The PUT statement is of the form:

```
PUT file <,record <,first>> ; expression list
```

where 'file' is the logical file number (1-127); and the 'expression list' contains 1 or more of the following entries:

numeric expression - writes 4 bytes containing the value of the expression;
string expression [byte count] - writes the specified number of bytes. The value of the string expression is truncated or blank filled on the right. The byte count range is 1-255.

Items between the angle brackets are optional and include: 'record', which is the record number to be written or updated (if 0 or omitted, then the record number is 1 greater than that used for the last access to the file); and 'first', which is the first byte of record to be written (1-record size). If no value is given, then first defaults to 1.

The PUT statement allows random records to be written or updated. For example:

```
PUT,1,R,13; "MORTGAGE COMPANY"[20]
```

updates 20 bytes of record R starting at the 13th byte.

8.7 Improving File Access

The random files in DISK BASIC are oriented towards fast random reads and updates. Sequential file input and output can easily be simulated; however, there is a time penalty for sequential output because the PUT statement updates information on a record. The file accessing time in a program can often be greatly reduced if the program takes advantage of the flexibility offered.

The file accessing scheme in DISK BASIC is different from the random accessing scheme commonly used in most microcomputers. When a record is accessed that is not present in one of the buffers in memory, the physical block containing the logical record is read into memory in an unused buffer or, if all buffers are in use, the least recently used (LRU) buffer. If the least recently used buffer has been modified, it is rewritten to disk before the next block is read into the buffer. This type of a buffer management scheme is very similar to the LRU virtual memory paging schemes used on large computers.

The first method of improving file access is by increasing the number of file buffers allocated in the FILE "R" statement. Changing this number from 1 to a larger number does not alter the results of execution; it only alters the number of times the disk has to be physically accessed. The difference in time can be quite substantial. However, for sequential access or random access which uniformly accesses all parts of a large file there is little advantage to be gained by increasing the number of buffers beyond 1.

The second method of improving file access is by varying the record size and blocking factor of a file. Ideally, the record size should be a power of 2. By choosing an appropriate blocking factor the block size will be a multiple of 128. For example, a 32 byte record can be blocked 4, 8, or 12, giving block sizes of 128, 256, or 384 bytes, respectively. For sequential access a blocking factor of 1 allocates 1 record to a physical block. Thus, to read records sequentially, 1 physical access and disk read is necessary for each record. With a blocking factor of 8, physical disk access is only necesary for every 8 records read, which is 1/8 as many disk accesses as necessitated by a blocking factor of 1.

If the record sizes are not a power of two, the blocking factor should be chosen carefully. For example, with 80 byte records a blocking factor of 1 will waste 48 bytes of disk space for each record because the 80 byte record is contained in a 128 byte disk sector. By using a blocking factor of 3, only 16 bytes (256-3*80) will be wasted for every 2 128 byte sectors. Again, with a blocking factor of 8, 640 bytes are used with no wasted space because 5 disk sectors hold exactly 640 bytes. Whether or not to choose 1, 3, or 8 should be determined by the type of application for which the file is used. If the program is large and uses most of the workspace, either 1 or 3 would be best. If the program is small, allocating 678 (34+4+640) bytes may be quite acceptable and improve the speed of the program. Choosing the best values for the number of buffers, record size, and blocking factor is often difficult. The user is following a reasonable guideline if he allocates 1 buffer for sequential files with a larger blocking factor and more buffers with smaller blocking factors for random files. For often used applications a little experimentation and fine tuning of the parameters can improve the disk access time.

8.8 Storage Requirements

When random files are used, they are allocated from the user's free workspace. The storage requirements in bytes are as follows:

error trapping - 10 bytes

open files -

4+30+BUF*(4+128*INT((RECSIZ*BLKFAC+127)/128)) bytes

where

BUF = the number of allocated physical block buffers,
RECSIZ = the number of bytes per record,
BLKFAC = the number of records per block.

Thus, opening a file with 80 byte records and a blocking factor of 3 and 1 buffer requires 34 + 1 * (4+256) = 294 bytes. With 4 buffers the requirement is 34 + 4 * (4+256) = 1074 bytes.

9. COLOR, GRAPHICS, AND OTHER TERMINAL FEATURES

9.1 The PLOT Statement

The PLOT Statement is used to output the 8 bit value of an expression to the screen. The form of the PLOT statement is as follows:

```
PLOT expression
```

or

```
PLOT expression,expression,...,expression
```

The expressions in the expression list must evaluate to a quantity in the range 0 to 255. Other values will cause a CF error.

For example, the following statement will cause the letters ABCDEF to be displayed on the screen.

```
PLOT 65,66,67,68,69,70
```

The PLOT statement is usually used to send control codes, escape codes, and other graphics information to the screen. For further examples, see the following sections in this chapter, and for information about CRT commands and ASCII codes, see Appendices C, E, and F.

9.2 Color

The color displays that can be achieved on the COMPUCOLOR II are an important feature of the machine. The color controls are easy to operate and add a new dimension to traditional programming.

Both the foreground and background can be set to a desired color. The foreground can be made to blink, and in addition, characters may be either single or double height.

Color, blink and character size can each be set in one of two ways. The first method involves the use of the color and special keys. To set the background color, the BG ON key is pressed. Then the actual color is set by simultaneously striking the CONTROL key and the letter key corresponding to the desired color. They are as follows:

BLACK:	P	BLUE:	T
RED:	Q	MAGENTA:	U
GREEN:	R	CYAN:	V
YELLOW:	S	WHITE:	W

On the deluxe and extended keyboards, the color keys are in a separate pad and are simply struck to select color.

The foreground can be set by depressing the FG ON key and selecting a color as for the background.

The BLINK ON key sets the blink in motion and the BL/A7 OFF key turns it off. The double-height characters can be set by the A7ON key and small characters are reset by the BL/A7 OFF key. Because this key controls both blink and character height, if the user wishes to turn the blink off while using the larger characters, and continue typing in large characters, the BL/A7 OFF key and the A7 ON key must be struck in immediate succession.

While these codes can be used in the CRT mode to test color combinatons and display appearances, etc., the characters will only be accepted in BASIC if they are contained in quoted strings or REMARK statements. If not so contained, they will cause a syntax error (SN).

Color can be selected without being contained in a quoted string by the second method of setting color, blink and character height. This is done through the use of the PLOT statement, as shown below:

PLOT 29	(sets foreground color)
PLOT 30	(sets background color)
PLOT 31	(sets blink on)
PLOT 14	(sets large characters)
PLOT 15	(sets blink and large characters off)

The individual colors are selected by PLOT statements using the internal code of each color key, as shown below:

PLOT 16	(black)	PLOT 20	(blue)
PLOT 17	(red)	PLOT 21	(magenta)
PLOT 18	(green)	PLOT 22	(cyan)
PLOT 19	(yellow)	PLOT 23	(white)

Because blink off and standard character height are controlled by the same code, retaining double character height while turning off the blink will require PLOT 15 and PLOT 14 statements in immediate sequence. The PLOT commands can be used in a BASIC program to set the color of the screen output.

The PLOT character set, BLINK, BACKGROUND COLOR, and FOREGROUND COLOR can also be selected by means of the CCI Control Code or PLOT 6 statement. The general form in BASIC is as shown:

PLOT 6,number

where number must be an integer between 0 and 255 representing the visible CCI status. This number is represented in binary up to eight bits long and arranged in a table as shown below. (Also shown in Appendix C.2)

A7	A6	A5	A4	A3	A2	A1	A0
PLOT	BLINK	BACKGROUND			FOREGROUND		
		BLUE	GREEN	RED	BLUE	GREEN	RED

The foreground and background colors are formed, as in a color television, by combinations of the blue, red, and green color guns. When the binary number is placed in the eight bit location, a 1 in any position turns that bit on. The formula for determining the desired number in decimal is:

```
PLOT*128 + BLINK*64 + BACKGROUND*8 + FOREGROUND
```

The program below illustrates the various results that can be achieved with the PLOT 6 command.

```
10 PLOT 6,6:REM SET CYAN FOREGROUND AND BLACK BACKGROUND
20 PRINT "PLOT(0-1),BLINK(0-1),BCKGRD(0-7),FORGRD(0-7): ";
25 INPUT "";PL,BL,BG,FG
30 PLOT 6,PL*128+BL*64+BG*8+FG
40 REM 30 SETS THE COLOR INFORMATION YOU SELECTED
50 PRINT "THIS IS WHAT YOU SELECTED";:PLOT6,6:PRINT
60 REM RESET COLOR BEFORE LINEFEED
70 GOTO 20
```

9.3 Special Characters

The COMPUCOLOR II has 64 special characters which are actually two groups of 32 special characters. A group is selected depending upon the condition of the Flag Bit. If the Flag Bit is off, then the ASCII codes from 96 to 127 are not changed when they are placed in the CRT refresh memory. If the Flag Bit is on, then these codes have 96 subtracted from them before they are replaced in the CRT refresh memory. Therefore, they are mapped into 0 to 31 within the refresh memory.

The characters in the range 96 to 127 are generated by changing the shift of the alphabetic characters @, A, ..., Z, [, \,], ^, and _. If the CAPS LOCK key is down, then the SHIFT key will also have to be depressed to generate these characters. If the CAPS LOCK key is up, then these special characters are generated whenever an alphabetic key is struck.

The condition of the Flag Bit is changed by depressing either the FG ON/FLAG OFF key or the BG ON/FLAG ON key. Thus, if the FG ON/FLAG OFF key is struck, the characters in columns 6 and 7 in the COMPUCOLOR II Character Set (shown in Appendix F) are displayed whenever ASCII codes in the range 96 to 127 ar received. If the BG ON/FLAG ON key is struck, the characters in columns 0 and 1 are selected.

In BASIC the two sets of special characters can be selected as follows:

PLOT CODE	KEY	CHARACTER SET
PLOT 29	FG ON/FLAG OFF	COLUMNS 6 AND 7
PLOT 30	BG ON/FLAG ON	COLUMNS 0 AND 1

The COMPUCOLOR II has an alternate set of 256 characters. These characters are used for the graphics plot modes where each character position is composed of eight plot blocks - four high by two wide. These plot blocks can also be accessed through the character plot via color pad mode entered by the ESC B sequence. This mode uses the eight color keys to intensify each of the eight plot blocks within a character. The one to one correspondence between the 4 x 2 color select pad on the extended and deluxe keyboards and the 4 x 2 character plot blocks is shown below.

CHARACTER PLOT BLOCK ARRAY WITH CORRESPONDING COLOR CODES

BLACK	BLUE
RED	MAGENTA
GREEN	CYAN
YELLOW	WHITE

This mode is designed especially for use by the keyboard to simplify the drawing of graphs or the correcting of graphs. Once this mode is entered a block at the top right hand corner of the present cursor position can be intensified by depressing the BLUE key at the top right hand corner of the color selection pad or CONTROL T (20).

Once a plot block is intensified, any other plot block in the same character positon can also be intensified since the cursor does not automatically advance. If a color key is corresponding to an intensified plot block is pressed, the plot block is turned off. This allows plot blocks to be erased. After all the desired plot blocks have been intensified or extinguished, the cursor can be moved using the cursor control keys without leaving this mode. In fact, all of the control codes are effective while in this mode except for the color select control codes, and any of the ASCII text characters (32 to 127) can be entered and displayed. Any code that requires a two key or more sequence (such as CURSOR X-Y, CCI, and ESC) terminates this mode. It should be noted that the ASCII text characters when entered and displayed advance the cursor. Therefore, when a character position has been used to display plot blocks, a cursor command must be given to advance the cursor to the next character positon.

9.4 Cursor Controls

The COMPUCOLOR II has two cursor modes available to the user. The most commonly used mode is the visible cursor mode where the blinking cursor on the screen shows the current visible cursor position. In this mode all the cursor control features of the COMPUCOLOR II are available. A second cursor mode, called the blind cursor mode, allows the use of a second invisible cursor with only XY cursor position allowed. The two modes are described in the following sections.

9.4.1 Visible Cursor Mode

In the visible cursor mode the following PLOT statements move the visible cursor on the screen:

PLOT 10	CURSOR DOWN / LINEFEED (moves the cursor 1 space down)
PLOT 25	CURSOR RIGHT (moves cursor one space to the right)
PLOT 28	CURSOR UP (moves cursor one space up)
PLOT 26	CURSOR LEFT (moves cursor one space to the left)
PLOT 8	HOME (moves cursor to topmost left of screen)
PLOT 9	TAB (moves cursor to start of next print zone)

The X,Y position of the visible cursor can be changed using the CURSOR X,Y control code sequence or the following PLOT statement:

```
PLOT 3,X,Y
```

where X and Y are the desired X,Y coordinates. The range for X is 0 to 64 and the range for Y is 0 to 31. Where X=0 and Y=0 is the HOME position at the upper left hand corner of the screen. The X coordinate determines the column position and the Y coordinate determines the line on the screen.

If the cursor is positioned at X=64, then the blinking visible cursor will disappear. But if a character is typed, it will be positioned at the beginning of the line specified by Y+1, and the cursor then reappears in character position (X=1). Any cursor movement command forces the cursor to reappear at the proper position relative to character position 0, line Y+1.

If the X value is greater than 64, then the blind cursor addressing mode is entered.

Page mode, which is entered from the keyboard via ESC X, writes characters left to right, and does not scroll the screen. From BASIC it is entered with a PLOT 27,24 statement.

Scroll mode, which is entered via ESC K, writes left to right and scrolls the screen for a continuous readout. It is entered in BASIC by PLOT 27,11.

Vertical mode, which is entered via ESC J writes top to bottom in one column only. It does not scroll the display. This mode can be reached through BASIC by the PLOT 27,12 statement.

ERASE PAGE code replaces the contents of the entire screen with spaces that have the same color and composite status as the present visible CCI status. Both the visible and blind cursors are positioned at HOME. In BASIC a PLOT 12 erases the screen.

The ERASE LINE code does a carriage return and replaces the line containing the visible cursor with spaces having the same color and composite status as the present visible CCI status. The cursor is sent to the beginning of the line. The current line is erased through BASIC by PLOT 11. The following program illustrates the use of some cursor controls.

```
10 DEF FNR(X) = INT (X*RND(1))
20 FOR I = 0 TO 3: READ D(I): NEXT I
30 DATA 10,25,28,26: REM CURSOR CONTROL VALUES
40 PLOT 6,0,12,27,24: REM ERASE PAGE AND SET PAGE MODE
50 PLOT 3, FNR(64), FNR(32): REM SELECT RANDOM STARTING POINT
60 FOR I = 1 TO 1000
70 PLOT 6, (FNR(7)+1)*8: REM SET VISIBLE BACKGROUND COLOR
80 PLOT 20,26: REM OUTPUT SPACE, THEN BACKSPACE
90 PLOT D(FNR(4)): REM OUTPUT A RANDOM DIRECTION
100 NEXT I
110 PLOT 6,2,8: REM SET COLOR AND RETURN HOME
120 END
```

9.4.2 Blind Cursor Mode

The blind cursor mode can be entered in two ways. The first is by using the CURSOR X,Y control sequence. If the X value is 65 or larger, then the terminal ignores this as the visible cursor X value and sends the unit into the blind cursor addressing mode. Once in the blind cursor X,Y addressing mode, three additional bytes must be sent. They are the blind cursor X value, the blind cursor Y value, and the blind status word. The blind X value must be in the range 0-63 and the blind Y value must be in the range 0-31. The blind status word has the same format as that required for the CCI code (PLOT 6) as described in Section 9.2.

The Blind A7 Bit will be set on by sending values from 128 to 255 instead of 65 to 127 when going from the visible cursor X,Y mode to the blind cursor X,Y mode. The Blind A7 Bit is set off when a value from 65 to 127 is used.

It should be noted that the X and Y cursor values are masked to 0-63 ad 0-31 respectively.

After receiving the five byte blind cursor X,Y sequence, the terminal is left in the blind cursor mode for whatever input device caused the mode to be entered. After CPU RESET, the keyboard and RS-232C serial port are placed in visible cursor mode. If the keyboard causes the blind cursor XY to be addressed, then the keyboard will be left in the blind cursor mode while the RS-232C serial is still in the visible cursor mode. This allows the keyboard and the RS-232C to use two different cursors.

It is important to note that most of the control codes affect only the visible cursor mode including all of the cursor positioning codes except, of course, CURSOR X,Y, which can affect both modes, and ERASE PAGE which resets both the visible and blind cursor to the home position (0,0). The setting of the Flag Bit is used by both the blind and visible cursor modes to select the proper special characters.

The blind cursor mode can also be entered by using the ESC A sequence, and ESC ESC returns the input to the visible cursor mode without changing the cursor address, composite color status word, or A7 Bit of the two cursor modes.

In BASIC the blind cursor X,Y addressing is used as follows:

```
PLOT 3, BC, X, Y, CL
```

where BC is in the range 65 to 127 for A7 OFF and small characters, and from 128 to 255 for A7 ON and large characters. X and Y are the cursor positions, and CL is the color status word (0-255). Blind cursor mode can be exited and visible cursor mode entered by:

PLOT 27,27 (ESC ESC)

and blind cursor mode can be re-entered by:

PLOT 27,1 (ESC A)

With the two cursor modes, the COMPUCOLOR II offers a great deal of flexibility as a color display device and a terminal. The blind cursor mode is useful for generating displays in character mode without the cursor interfering with the display. For example, the differences are shown by the hunter and turkey program below:

```
0 REM TURKEY AND THE HUNTER
10 REM VISIBLE AND BLIND CURSOR DEMONSTRATION
20 PLOT 6,2,12:INPUT "VISIBLE OR BLIND CURSOR (V/B)?";A$
30 BC=MID$(A$,1,1)="B":VC=MID$(A$,1,1)="V"
40 IF BC+VC<>-1 THEN 20
50 REM DRAW BORDER AROUND SCREEN
60 PLOT 27,24:REM PAGE MODE
61 PLOT 15:REM A7 OFF - SMALL CHARACTERS
62 PLOT 6,0:REM SET COLOR - BLACK FG/BLACK BG
63 PLOT 12:REM ERASE PAGE
70 PLOT 6,15:REM SET COLOR - WHITE FG/RED BG
71 FOR I=1 TO 64:PLOT 32:NEXT:REM DRAW TOP LINE
72 PLOT 3,0,31:REM MOVE CURSOR TO BOTTOM LINE
73 FOR I=1 TO 64: PLOT 32:NEXT:REM DRAW BOTTOM LINE
74 PLOT 27,10:REM WRITE VERTICAL MODE
75 PLOT 8:REM MOVE CURSOR TO HOME
76 FOR I=1 TO 32: PLOT 32:NEXT:REM DRAW LEFT SIDE
77 PLOT 3,63,0:REM MOVE CURSOR TO TOP RIGHT
78 FOR I=1 TO 32:PLOT 32:NEXT:REM DRAW RIGHT SIDE
79 PLOT 27,24:REM PAGE MODE
90 PLOT 3,64,0:REM MOVE BLINKING CURSOR OFF SCREEN
100 REM SET UP GAME PARAMETERS
110 HX=1: HY=1: REM HUNTER INITIAL POSITION
120 TX=32:TY=16: REM TURKEY INITIAL POSITION
130 TS=2:REM TURKEY SPEED
150 HC=39:TC=15
180 REM DEFINE FNR TO RETURN RANDOM INTEGER IN RANGE -X TO X
190 DEF FN R(X)= -X+INT((2*X+1)*RND(1))
200 REM MOVE CURSOR TO TURKEY'S OLD POSITION
201 IF VC THEN PLOT 3,TX,TY,6,0: REM VISIBLE
202 IF BC THEN PLOT 3,127,TX,TY,0: REM BLIND
210 TX=TX+FNR(TS):REM CHANGE TURKEY X POSITION
220 TY=TY+FNR(TS):REM CHANGE TURKEY Y POSITION
230 IF TX<1 OR TX>62 OR TY<1 OR TY>30 THEN 1000:REM ESCAPE!
240 PLOT 32:REM CLEAR TURKEY'S LAST POSITION
250 REM MOVE CURSOR TO TURKEY'S NEW POSITION
```

```
251 IF VC THEN PLOT 3,TX,TY,6,TC:REM VISIBLE
252 IF BC THEN PLOT3,127,TX,TY,TC:REM BLIND
260 PLOT ASC("T"):REM OUTPUT TURKEY SYMBOL
300 REM MOVE CURSOR TO HUNTER'S OLD POSITION
301 IF VC THEN PLOT 3,HX,HY,6,0:REM VISIBLE
302 IF BC THEN PLOT 3,127,HX,HY,0:REM BLIND
310 REM RANDOM SELECT HUNTER'S MOVE IN X OR Y DIRECTION
320 IF  RND(1)>(ABS(TY-HY)+1)/(ABS(TY-HY)+ABS(TX-HX)+2)THEN500
400 HY=HX+SGN(TY-HY):REM MOVE TOWARD TURKEY IN Y DIRECTION
410 GOTO 600
500 HX=HX+SGN(TX-HX): REM MOVE TOWARD TURKEY IN X DIRECTION
600 PLOT 32:REM CLEAR HUNTER'S LAST POSITION
700 REM MOVE CURSOR TO HUNTER'S LAST POSITION
701 IF VC THEN PLOT 3,HX,HY,6,HC: REM VISIBLE
702 IF BC THEN PLOT 3,127,HX,HY,HC: REM BLIND
710 PLOT ASC("H"): REM OUTPUT HUNTER SYMBOL
720 IF HX=TX AND HY=TY THEN 2000:REM HUNTER CATCHES TURKEY
800 GOTO 200
1000 REM TURKEY ESCAPES
1010 PLOT 27,27:REM VISIBLE CURSOR MODE
1020 PLOT 6,2: REM SET COLOR - GREEN FG/BLACK BG
1030 PLOT 8:REM CURSOR HOME
1040 PRINT "TURKEY ESCAPES !!!!   "
1050 GOTO 3000
2000 REM HUNTER CATCHES TURKEY
2010 PLOT 27,27:REM BLIND CURSOR MODE
2020 PLOT 6,2:REM SET COLOR - GREEN FG/BLACK BG
2030 PLOT 8:REM CURSOR HOME
2040 PRINT "GOBBLE GOBBLE .....  "
3000 FOR I=1 TO 1000:NEXT : REM DELAY FOR A WHILE
3010 RUN
```

9.5 Vector Graphics

The vector graphics capability of the COMPUCOLOR II allows the user to draw almost any desired display. The vector graphics are enabled by entering the graphic plot mode by depressing CONTROL B (binary 2) from the keyboard when in the CRT mode, or by executing PLOT 2 in BASIC. While in the graphic plot submode the user can choose from sixteen (16) plot submodes that perform a variety of graphic functions. The initial plot submode is the XY Point Plot mode. In this mode the user can turn on and off individual plot blocks on the screen. Other plot submodes can easily be entered by a binary code from 240 to 255.

An additional feature is available to allow a graphic plot to be erased by simply setting the FLAG bit on before entering the plot mode. This causes a logical XOR function to be used in setting the plot blocks. Thus, if the same point is plotted a second time, it is erased. Also, any plot submode may be entered from any other plot submode except Character Plot mode. The various submodes and their interactions are explained in detail below.

Colors may be defined on a character by character basis only and the color of an individual plot block as well as other intensified plot blocks within a character will be the most recent color defined when a new plot block within that character is turned on. To change color, it is necessary to exit the current plot submode, set the new color, and re-enter the plot mode.

The character grid on the screen is 64 characters wide and 32 characters high. The zero reference point for all plotting is the lower left hand corner of the screen. Each character is further subdivided into 8 plot blocks -- 2 blocks wide and 4 blocks high. This gives a 128 by 128 grid of plot blocks which may be individually set. All plot submodes operate on this grid size and have the same reference point (0,0). Positive directions are up and to the right, and negative directions are down and to the left.

All plot submodes and the general Plot Mode are terminated or exited by the binary code 255. When ever this code is issued, the plot mode is terminated and must be re-entered by issuing a CONTROL B or binary 2.

On the deluxe keyboards there are sixteen (16) special functions keys labelled F0 through F15. Using these keys the various plot submodes can be entered directly in the CRT mode (not in BASIC.) The F0 key produces a binary 240 code, F1 a 241, etc., up to the F15 key which produces a 255. In BASIC these plot submodes are entered by using the PLOT statement as described below.

Plot Mode Escape - (255 binary)

This code is used to exit from the Plot Mode or any of the plot submodes. On the deluxe keyboards the F15 function key performs a Plot Mode Escape.

Character Plot - (254 binary)

The Character Plot Submode is entered by a 254 after the general Plot Mode is entered. All subsequent characters issued are treated as plot characters except for 255 which is the Plot Mode Escape. Thus, other plot submodes can not be entered directly from this mode. The plot characters are constructed by ORing together the selected plot blocks to form the composite character as follows:

01 HEX	1 0 0 0 0 0 0 0	10 HEX	0 1 0 0 0 0 0 0
02 HEX	0 0 1 0 0 0 0 0	20 HEX	0 0 0 1 0 0 0 0
04 HEX	0 0 0 0 1 0 0 0	40 HEX	0 0 0 0 0 1 0 0

08 HEX	0 0 0 0 0 0 1 0	80 HEX	0 0 0 0 0 0 0 1

The Character Plot causes the the 6 wide by 8 high dot matrix to be divided into 8 blocks organized 2 blocks wide and 4 blocks high. Each block consists of a dot matrix 3 dots wide and 2 dots high. Each block corresponds to an individual bit of the 8 bit plot character. Large characters may also be formed by using the plot blocks in several character positions to create a large 5 by 7 matrix or any other desired size.

X Point Plot - (binary 253)

The X Point Plot is automatically entered upon receipt of the general Plot Mode code, binary 2 or CONTROL B. It may also be entered directly from any of the other plot submodes. After entering the X Point Plot submode, the next byte received defines the X value of the block that is desired to be plotted. The X value may range from 0 to 127 and all other values will cause 128 to be subtracted from the value of X.

The X Point Plot may be terminated by the code 255 which also causes the the general Plot Mode to be terminated. Any of the other plot submodes may be entered directly from the X Point Plot by simply entering the appropriate plot submode codes from 240 to 255.

It should be noted that this plot submode does not cause a plot block to be intensified, it only defines the X value. Once the X value is received, the COMPUCOLOR II is automatically placed in the Y Point Plot mode. Thus, the next code sent will be the Y value which may range from 0 to 127.

The procedure for entering and exiting the X Point Plot mode is shown below:

Function	Code
Plot Mode	2
X1 Value	0 to 127
Y1 Value	0 to 127
.	.
.	.
.	.
Xn Value	0 to 127
Yn Value	0 to 127
Plot Escape	255
or	
Plot Submode	240 to 254

The X Point Plot in conjunction with the Y Point Plot allows any block on a 128 by 128 block matrix to be intensified. Thus, in BASIC the above sequence becomes:

```
PLOT 2,X1,Y1, ... ,XN,YN,255
```

The following statement will plot points at the screen's four corners:

```
PLOT 2, 0,0, 0,127, 127,127, 127,0, 255
```

Y Point Plot - (binary 252)

The Y Point Plot is entered by a binary 252 code after the general Plot Mode is entered or automatically from the X Point Plot submode after the X value has been sent. The next byte received after entering the Y Point Plot submode defines the Y value of the block to be plotted and intensifies that block. If the new block is within a character position that contains an ASCII character, then the ASCII character is replaced completely by the new block and its associated color.

XY Incremental Point Plot - (binary 251)

The XY Incremental Point Plot submode is entered by a binary 251 code while in the general Plot Mode. The next byte defines the next two (2) increments as shown below. This byte may take on values in the range 0 to 239 since the binary codes from 240 to 255 are used for the plot submodes.

```
b7  b6  b5  b4  b3  b2  b1  b0

[ X  ]  [ Y  ]  [ X  ]  [ Y  ]
   1       1       2       2

Plot Block 1    Plot Block 2
```

The 4 two bit codes are defined as follows:

0	No change
1	Negative increment
2	Positive increment
3	No change

If b0 through b3 are "0"s, then the plot block will not plot, but will still increment according to the coding of b4 through b7. This allows skipping a plot increment by plotting an "invisible" block. The XY Incremental Plot mode may be terminated by the Plot Mode Escape code 255. The following sample program will do a random walk using the Incremental Point Plot mode.

```
10 DEF FNR(X)=INT(X*RND(1))
20 PLOT 12,6,6 : REM CLEAR SCREEN AND PLOT IN LIGHT BLUE
30 PLOT 2,63,63 : REM PLOT POINT IN THE MIDDLE OF THE SCREEN
40 PLOT 251 : REM ENTER INCREMENTAL POINT PLOT MODE
50 FOR I=1 TO 1000
60 INC=FNR(3)*64+FNR(3)*16+FNR(3)*4+FNR(3)
70 REM  USE ONLY THE FIRST THREE DIRECTION CODES
75 IF (INC AND 15)=0 THEN 60 :REM NO ALLOW INVISIBLE BLOCKS
80 PLOT INC
90 NEXT I
100 PLOT 255 : REM ESCAPE FROM PLOT MODE
110 END
```

X Bar Graph, X0 Value - (250 binary)

The X Bar Graph, X0 Value plot submode is entered by a binary 250 code after the general Plot Mode is entered. It may also be entered directly from any of the other plot submodes except for Character Plot. After entering the X Bar Graph, X0 Value submode, the next byte defines the X0 value or the left horizontal start block of the horizontal bar graph. The X0 may range in value from 0 to 127 and all other values have 128 subtracted giving a new X0 value in the range 0 to 127.

Upon receiving the X0 value, the value of X0 is stored in memory and the COMPUCOLOR II is automatically placed in the X Bar Graph, Y Value plot submode (249 binary.) After receiving the next byte as the Y value, the COMPUCOLOR II is automatically placed in the X Bar Graph, X Max Value plot submode (248 binary.) After receiving the X Max value the horizontal bar graph is drawn on the screen and the COMPUCOLOR II is placed back in the X Bar Graph, Y Value plot submode ready to receive new Y and X Max value pairs until a new plot submode is entered. Note that once an X0 value is defined it is unnecessary to respecify it for each horizontal bar in the graph. This process is shown in the following example.

Function	Code
Plot Mode	2
or	
Plot Submode	240 to 253
X Bar Graph, X0 Value	250
X0 value	0 to 127
Y value - line 1	0 to 127
X Max value - line 1	0 to 127
.	.
.	.
.	.
Y value - line n	0 to 127
X Max value - line n	0 to 127
Plot Escape	255
or	
Plot Submode	240 to 254

For example, from BASIC a horizontal bar graph plotting a sine function can be drawn as follows:

```
10 PLOT 6,6,12   :REM SET COLOR TO CYAN AND CLEAR SCREEN
20 X0 = 10       :REM SET X0 VALUE
30 PLOT 2,250,X0:REM ENTER X BAR GRAPH SUBMODE - SET X0
40 FOR Y=0 TO 127 STEP 2  :REM SET Y VALUES
50 PLOT Y,X0+50*(1+SIN(Y/10)) :REM SCALE SINE FUNCTION
60 NEXT Y
70 PLOT 255      :REM PLOT ESCAPE
```

As can be seen from the above examples, once in the X Bar Graph, X0 mode, it is necessary only to define only two points for each new bar graph. The bar graph is drawn after receiving the X Max value. Any of the other plot submodes can be entered directly from the three X Bar Graph submodes. Multiple colored bar graphs can be drawn by leaving plot mode, changing the color, and re-entering the X Bar Graph, Y Value submode (249 binary.) In this case the original XO value would be preserved. Bars drawn in this mode are one plot block wide; thicker bars can be drawn by changing the Y value by 1 and replotting it along with the same X Max value or using the X Incremental Bar Graph submode.

X Bar Graph, Y Value - (249 binary)

The X Bar Graph, Y Value plot submode is entered by a binary 249 code or automatically from the X Bar Graph, XO Value plot submode. After entering this submode the next byte is used as the Y value of the next bar in the graph to be plotted, and the COMPUCOLOR II is automatically placed into the X Bar Graph, X Max Value plot submode (248 binary.) Any of the other plot submodes can be entered directly from this submode. For more information on this submode see the description of the X Bar Graph, XO Value submode (250 binary.)

X Bar Graph, X Max Value - (248 binary)

The X Bar Graph, X Max Value plot submode is entered by a binary 248 code or automatically from the X Bar Graph, Y Value plot submode. After entering this submode the next byte is used as the X Max value of the bar in the graph. The bar is plotted, and the COMPUCOLOR II is automatically placed into the X Bar Graph, Y Value plot submode (249 binary) which allows the next bar to be defined and drawn. Any of the other plot submodes can be entered directly from this submode. For more information on this submode see the description of the X Bar Graph, XO value submode (250 binary.)

X Incremental Bar Graph - (247 binary)

The X Incremental Bar Graph plot submode is entered by a binary 247 code. After entering this submode the next byte defines the next two horizontal and vertical increments for two horizontal bar graphs. Thus, it is possible to position a bar graph on either side of the present location by adding or subtracting an increment to the bar graph previously defined. The coding and composition of the incremental direction code is the same as that defined in the XY Incremental Point Plot submode (251 binary.) Any of the other plot submodes can be entered directly from this submode.

Y Bar Graph, YO Value - (246 binary)

The Y Bar Graph, YO Value plot submode is entered by a binary 246 code after the general Plot Mode is entered. It may also be entered directly from any of the other plot submodes except for Character Plot. After entering the Y Bar Graph, YO Value submode, the next byte defines the YO value or the bottom vertical start block of the vertical bar graph. The YO may range in value from 0 to 127 and all other values have 128 subtracted giving a new YO value in the range 0 to 127.

Upon receiving the Y0 value, the value of Y0 is stored in memory and the COMPUCOLOR II is automatically placed in the Y Bar Graph, X Value plot submode (245 binary.) After receiving the next byte as the X value, the COMPUCOLOR II is automatically placed in the Y Bar Graph, Y Max Value plot submode (244 binary.) After receiving the Y Max value the vertical bar graph is drawn on the screen and the COMPUCOLOR II is placed back in the Y Bar Graph, X Value plot submode ready to receive new X and Y Max value pairs until a new plot submode is entered. Note that once a Y0 value is defined, it need not be respecified for each vertical bar in the graph. This is shown in the following example.

Function	Code
Plot Mode	2
or	
Plot Submode	240 to 253
Y Bar Graph, Y0 Value	246
Y0 value	0 to 127
X value - line 1	0 to 127
Y Max value - line 1	0 to 127
.	.
.	.
.	.
X value - line n	0 to 127
Y Max value - line n	0 to 127
Plot Escape	255
or	
Plot Submode	240 to 254

For example, from BASIC a vertical bar graph plotting the area under a random function can be drawn as follows:

```
10 PLOT 6,6,12  :REM SET COLOR TO CYAN AND CLEAR SCREEN
20 Y0 = 10      :REM SET Y0 VALUE
30 PLOT 2,246,Y0:REM ENTER Y BAR GRAPH SUBMODE - SET Y0
40 FOR X=0 TO 127 STEP 2:REM SET X VALUES
50 PLOT X,Y0+100*RND(1)  :REM SCALE RANDOM FUNCTION
60 NEXT X
70 PLOT 255     :REM PLOT ESCAPE
```

As can be seen from the above examples, once in the Y Bar Graph, Y0 mode, it is necessary to define only two points for each new bar in the graph. The bar graph is drawn after receiving the Y Max value. Any of the other plot submodes can be entered directly from the three Y Bar Graph submodes. Multiple colored bar graphs can be drawn by leaving plot mode, changing the color, and re-entering the Y Bar Graph, X Value submode (245 binary.) In this case the original Y0 value is preserved. Bars drawn in this mode are one plot block wide; thicker bars can be drawn by changing the X value by 1 and replotting it along with the same Y Max value or using the Y Incremental Bar Graph submode.

Y Bar Graph, X Value - (245 binary)

The Y Bar Graph, X Value plot submode is entered by a binary 245 code or automatically from the Y Bar Graph, Y0 Value plot submode. After entering this submode the next byte is used as the X value of the next bar to be plotted, and the COMPUCOLOR II is automatically placed into the Y Bar Graph, Y Max Value plot submode (244 binary.) Any of the other plot submodes can be entered directly from this submode. For more information on this submode see the description of the Y Bar Graph, Y0 Value submode (246 binary.)

Y Bar Graph, Y Max Value - (244 binary)

The Y Bar Graph, Y Max Value plot submode is entered by a binary 244 code or automatically from the Y Bar Graph, X Value plot submode. After entering this submode the next byte is used as the Y Max value of the bar in the graph. The bar is plotted, and the COMPUCOLOR II is automatically placed into the Y Bar Graph, X Value plot submode (245 binary) which allows the next bar to be defined and drawn. Any of the other plot submodes can be entered directly from this submode. For more information on this submode see the description of the Y Bar Graph, Y0 value submode (246 binary.)

Y Incremental Bar Graph - (243 binary)

The Y Incremental Bar Graph plot submode is entered by a binary 243 code. After entering this submode the next byte defines the next two vertical and horizontal increments for two vertical bar graphs. Thus, it is possible to position a bar graph on either side of the present location by adding or subtracting an increment to the bar graph previously defined. The coding and composition of the incremental direction code is the same as that defined in the XY Incremental Point Plot submode (251 binary.) Any of the other plot submodes can be entered directly from this submode.

X0 Vector Plot - (242 binary)

The X0 Vector Plot submode is entered by a binary 242 code after the general Plot Mode is entered. After entering the X0 Vector Mode the next byte defines the X0 point of the vector being drawn. The vector mode requires two endpoints to be defined (i.e. X0,Y0 and X1,Y1.) The X1,Y1 values should be previously defined by way of the X and Y Point Plot submodes (253 and 252 binary.) Upon receiving the X0 value the COMPUCOLOR II is automatically placed into Y0 Vector Plot submode. After receiving the Y0 value the COMPUCOLOR II plots the best fitting straight line between X0,Y0 and X1,Y1 using the plot blocks and returns to the X0 Vector Plot submode, ready to plot vectors between successive X0,Y0 pairs. This process is shown below:

Function	Code
Plot Mode	2
or	
X Point Plot	253
X1 Vector point 1	0 to 127
Y1 Vector point 1	0 to 127
X0 Vector Plot	242
X0 Vector point 1	0 to 127
Y0 Vector point 1	0 to 127
.	.
.	.
.	.
X0 Vector point n	0 to 127
Y0 Vector point n	0 to 127
Plot Escape	255
or	
Plot Submode	240 to 254

Thus, in BASIC the above sequence becomes

```
100 PLOT 2, X1,Y1
110 PLOT 242
120 FOR I=1 TO N
130 PLOT X0(I),Y0(I)
140 NEXT I
150 PLOT 255
```

To plot a rectangle around the entire screen simply execute the statement

```
PLOT 2, 0,0, 242, 0,127, 127,127, 127,0, 0,0, 255
```

Y0 Vector Plot - (241 binary)

The Y0 Vector Plot submode is entered by a binary 241 code after the general Plot Mode is entered. After entering this submode the next byte defines the Y0 value of the vector being drawn. There is no restriction on Y0 except that it must be in the range 0 to 127. Upon receiving the Y0 value a vector is plotted from X1,Y1 to X0,Y0 with X0,Y0 replacing the old X1,Y1 endpoint. If the next vector has a X1,Y1 value equal to the old X0,Y0 value, then only the new X0,Y0 values need be sent. This effectively draws a vector from the present X0,Y0 position to the new X0,Y0 position. For more information on this submode see the description of the X0 Vector Plot submode (242 binary.)

Incremental Vector Plot - (240 binary)

The Incremental Vector Plot submode is entered by a binary 240 code after the general Plot Mode is entered. After entering this submode the next byte defines the increments in the X0,Y0 and X1,Y1 values for the vector from X1,Y1 to X0,Y0. The values for the increments are defined as follows:

```
b7  b6  b5  b4  b3  b2  b1  b0

[ X   ] [ Y   ] [ X   ] [ Y   ]
   1       1       0       0
```

The 4 two bit codes for the increments are defined as follows:

0	No change
1	Negative increment
2	Positive increment
3	No change

The incremental direction codes are similar to those used for the other increment plot submodes. Furthermore, if either half of the word is all zeroes, then the corresponding X,Y values will be changed but no vector will be drawn. This allows endpoints for the vectors to be skipped. The only time a vector is drawn is when both halfs of the word are non-zero. The Incremental Vector Plot submode does not automatically transfer control to any other plot submode. Therefore, a series of incremental movements in both X1,Y1 and X0,Y0 can be made by sending consecutive incremental direction codes.

9.6 RS-232C Interface

The RS-232C interface allows the user to connect any RS-232C compatible device to the COMPUCOLOR II. For instance, this enables most serial printers to be interfaced without any additional software.

The RS-232C port is controlled by several escape codes which set the baud rate of the serial output and direct all output to the serial port. The default baud rate of the serial port is 9600 baud with 1 stop bit. This rate can be changed by using the ESC R sequence. The setting of the A7 Bit determines the number of stop bits when the ESC R sequence is given. A7 OFF gives 2 stop bits (normal for 110 baud) and A7 ON gives 1 stop bit. The baud rate is selected by issuing the sequence ESC R followed by a character in the range 1 through 7 which specifies the baud rate as shown in the table below.

BAUD RATE SELECTION

NUMBER KEY	1	2	3	4	5	6	7
BAUD RATE	110	150	300	1200	2400	4800	9600

After the baud rate has been properly set, data can be transmitted to the RS-232C serial port by executing an ESC M sequence. Once this escape sequence has been issued, all inputs to the CRT display drives (including all keyboard inputs and BASIC outputs) are directed to the RS-232C port instead of the CRT screen. Thus, the only way to break out of this mode from the CRT mode is via the CPU RESET key or in Disk BASIC by executing a POKE 33265,0 statement which resets the BASIC Output Flag to send characters to the seen in visible cursor mode. In BASIC the COMPUCOLOR II can be reset to the previous output mode by saving the contents of the BASIC Output Flag with an X=PEEK(33265) function before issuing an ESC M sequence and then restoring the BASIC Output Flag by executing the statement POKE 33251,X as follows:

```
10 REM SET BAUD RATE = 300, 1 STOP BIT
20 PLOT 14,27,18,3,15
100 GOSUB 9000: REM DIRECT OUTPUT TO RS-232C PORT
110 FOR I=1 TO 10
120 PRINT "THIS IS AN RS-232C TEST"
130 NEXT
140 GOSUB 9500: REM RESET OUTPUT TO CRT
150 PRINT "BACK TO THE CRT"
160 END
9000 TMP = PEEK(33265): REM SAVE BASIC OUTPUT FLAG
9010 PLOT 27,13: REM OUTPUT TO RS-232C
9020 RETURN
9500 POKE 33265, TMP:REM RESET OUTPUT TO CRT
9510 RETURN
```

Several other control and escape codes interact with the RS-232C serial port. The ESC C sequence transmits the cursor and color status to the RS-232C port using the following sequence:

3, X, Y, 6, Status, ASCII Character, 13

The X,Y values are the current cursor position, Status is the color status word of the ASCII Character stored at the cursor position. Using the CONTROL X code, every text character on the screen is transmitted to the RS-232C port from the visible cursor to the end of the page or until an FF,00 sequence is found in the screen refresh RAM. The text characters are sent in lines terminated by a linefeed and carriage return. The color status is not transmitted.

NOTE - When interfacing any device to the RS-232C serial port it may be necessary to switch the transmit and receive data lines at device end of the cable (lines 2 and 3). See Appendix D.4 for pin assignments on the COMPUCOLOR II.

9.7 Using the COMPUCOLOR II as a Terminal

The COMPUCOLOR II can be used as a data communications terminal with the RS-232C interface. To enter the terminal mode, the user must strike CPU RESET, which places the COMPUCOLOR II into CRT mode or terminal mode.

Initially, the CRT is placed in local mode where all keyboard inputs are echoed to the screen. At this point the correct baud rate should be set. Half-duplex mode and full-duplex mode may be entered by striking the ESC H and ESC F sequences, respectively. In half duplex mode all keyboard inputs are echoed both to the CRT and to the RS-232C port. In full duplex mode all keyboard inputs are directed only to the RS-232C port. By striking the BREAK key, full duplex mode can be exited and half duplex mode entered.

NOTE - When interfacing a modem or acoustic coupler to the RS-232C serial port, it usually is not necessary to switch the transmit and receive data lines because the RS-232C port is configured as if the COMPUCOLOR II were a data communications terminal. On some computer systems using 7 bit ASCII codes, correct parity is required. The COMPUCOLOR II is set up to transmit and receive 8 bit ASCII characters and data which precludes parity checking. In this case proper communications will require a special communications program that executes in the user's RAM workspace.

Another problem that may be encountered is due to the fact that the COMPUCOLOR II responds to almost every ASCII control code and escape sequence. If the host computer sends control codes other than NULL, CARRIAGE RETURN, LINEFEED, and ERASE PAGE, then the COMPUCOLOR II may respond in an unexpected fashion. In these cases it is probably best if the host computer treats the COMPUCOLOR II as a TELETYPE instead of a CRT dislay terminal.

9.8 Miscellaneous Escape Codes

The COMPUCOLOR II has several additional escape codes to test the display, and jump to fixed and user defined memory locations. A test pattern can be generated on the CRT screen using the ESC Y test mode sequence. By issuing an ESC Y followed by a character, the entire screen is filled with that character using the current visible status word as the visible status word for each character on the CRT screen.

Several of the remaining ESCAPE codes have been pre-programmed to execute JMP's to certain memory locations as outlined below.

ESCAPE CODE	MEMORY LOCATION HEX	DECIMAL
I	9000	36864
S	A000	40960
T	8200	33280
^	81BF	33215

The ESC ^ is a user definable escape code. By POKEing an 8080 JMP instruction into the three bytes starting at 33215 a jump to any location in memory can be defined.

10. FCS

10.1 Introduction to FCS

The File Control System, or FCS, is used to manage the diskettes which store programs. The File Control System enables the user to store and save programs, screen displays, and arrays.

To enter FCS the user must first type ESC D, then the message prompt FCS> will appear. Once in the File Control System, commands should be entered after the FCS> prompt. For example, the command DIR should be used for listing the directory of a diskette. To change from one drive to another, the command DEV0: must be typed for the internal drive, and DEV1: must be typed for the external disk drive.
Machine-code programs may be in either one of two different FCS file types:

FILE TYPE .PRG

A .PRG type file is created with the FCS SAVE command. It is a machine-code program in "Memory image" form. The information in the file is a contiguous memory image of the program. The RUN command will load a .PRG file into memory starting at the specified Load Address in the file's directory entry, and begin execution at the Start Address specified in the file's directory entry. A .PRG file is loaded into memory much faster than an .LDA file. Therefore, once a program is working, it should be saved in .PRG form with the SAVE command, so that subsequent RUN's of the program will be quicker.

FILE TYPE .LDA

An .LDA type file is created by the COMPUCOLOR 8080 Assembler. The file consists of one or more data records and is terminated by one end record. Each data record specifies a load address for the record, and one or more data bytes to be loaded sequentially into memory starting at the load address. The end record specifies the starting (execution) address for the program (the operand of the END statement in the source program).

FCS can also be entered via the ESC G sequence. In this mode all outputs are sent to the RS-232C serial port and the prompts and inputs are echoed to the CRT screen. The ESC ESC sequence will exit FCS.

In BASIC FCS can be called by issuing a PLOT 27,4 (or PLOT 27,7 for output to the RS-232C port). All subsequent outputs from BASIC are treated as inputs to FCS. This mode is exited by executing a PLOT 27,27 statement which returns control of BASIC's output back to BASIC. See Section 8.2 for further details.

10.2 The FCS Commands

The FCS system has a number of commands which enable the user to manipulate records as desired. A list of commands appears in Appendix

B.1. The following commands are used as explained below. Before any of these commands may be used, the user must first enter the File Control System by typing ESC D as described above. In the following descriptions of commands, angle brackets, <>, will be used to denote an element of a statement that is optional. The 'Device Name' refers to the name and number of the disk drive being used. The COMPUCOLOR II has an internal disk drive, CD0, and an optional external disk drive, CD1. The 'File Spec' is the name that the user has assigned to the file followed by the file type (.PRG, .LDA, .BAS, etc.) and, optionally, a semicolon (;) followed by a version number in the range 01 to FF HEX. If the specified file is being read, then the default version is the file with the largest version number. With files being written, the default version number is one higher than the largest version number of an existing file on the specified device. If no file currently exists on the disk with the specified name, then the default version number is 01. The 'Memory Spec' is the 'Start Address' in HEX followed by the number of bytes or followed by hyphen (-) and the 'End Address'. Only the first three letters of any command are required.

CAUTION: When a COPY, DELETE, or DUPLICATE command is executed, the screen memory is used, and any screen display will be lost. A brief character display will appear during the exucution of these commands.

COPY

The COPY command allows the user to copy a file, possibly to another disk drive, and is of the form:

```
COPY <Device Name:> File Spec TO <Device Name:> File Spec
```

For example:

```
COP 0:TEST.PRG TO 1:ABC
```

When entered, this command will copy the latest version of TEST.PRG on device 0 to file name ABC.PRG on device 1. The COPY command used the screen memory as a temporary buffer.

DELETE

The DELETE command allows for the deletion of any file on the diskette, and is of the form:

```
DELETE <Device Name:> File Spec
```

For example:

```
DEL TEST.BAS;1
DEL 1:TEST.PRG;2
DEL CD1:NAME.RND;1
```

The complete File Spec is needed to delete a file. This form of file protection is provided to prevent accidental erasures. The DELETE command repacks the disk and directory by using the screen memory as a temporary buffer.

DEVICE

The DEVICE command allows the user to change the default device or drive, and is of the form:

DEVICE <Device Name:>

If the Device Name is not specified, then the current default device is listed. For example:

DEV CD0:

will change the default device to the COMPUCOLOR II internal disk drive.

DIRECTORY

The DIRECTORY command lists all the programs on the diskette on any device, and is of the form:

DIRECTORY <Device Name:>

For example:

DIR
DIR CD1:

A directory listing may be halted by striking the BREAK key, and it may be resumed by striking the RETURN key. If the LINEFEED key is struck after the BREAK key, the directory is stopped and the machine is ready to receive another command.

DUPLICATE

The DUPLICATE command allows all the files on one diskette to be copied to another diskette. The two specified devices must be of the same type, but have different numbers. The command is of the form:

DUPLICATE Device Name: TO Device Name:

For example:

DUP 0: TO 1:

The DUPLICATE command uses the screen memory as a temporary buffer.

INITIALIZE

The INITIALIZE command allows the user to give a diskette a ten-letter name and optionally assign the number of allotted directory blocks. This command clears all the directory information on a diskette, effectively deleting all files on the diskette. It should only be used when a "clean" diskette is desired. It is of the form:

    INITIALIZE <Device Name:> Volume Name No. of DIR blocks

For example:

    INI CD0:SAMPLENAME
    INI CD1:TESTDISK01 10 (the 10 is optional)

The COMPUCOLOR II Disk directory size defaults to 6 blocks which can hold 34 files. Each directory block can hold information on 6 files; however, 2 entries are neccesary for the Volume Name and free space entries, i.e. 34 = 6 * 6 - 2.

LOAD

The LOAD command allows the user to load any type file into any RAM memory location he may wish. This indicates that the user may bring a display to the screen which is correct. LOAD command uses the same guide lines as the SAVE command. The LOAD command operates differently depending on the file type loaded. The default type is .LDA.

To LOAD a file type other than .LDA, the command is of the form:

    LOAD <Device Name:> File Spec <Load Address>

The file is assumed to be a "memory image" file and is loaded contiguously into memory starting either at the load address in the file's directory entry or at the load adress specified in the command line.

To load a file of type .LDA, the command is of the form:

    LOAD <Device Name:> File Spec <Lowest Address <Memory Spec>>

Each data record in the file is loaded into memory. If Lowest Address and Memory Spec are not specified, then each record is loaded at the address specified in the record.

If Lowest Address and Memory Spec are specified, the default Memory Spec is A000-FFFF. A "memory range" will be determined as follows:

1. If the Memory Spec is omitted, the range will be A000-FFFF.
2. If one number, i.e. C000, is given for the Memory Spec, then the range will be specified by the given number as the low limit and FFFF as the high limit of the range.
3. If two numbers, separated by a hyphen are given for the Memory Spec, then the range is specified by those numbers.
4. If two numbers, separated by a space or comma, are given for the Memory Spec, then the first number will be the low limit

of the range, and the second number is the byte count used to calculate the high limit of the range. For example, D000 400 will give a range D000-D3FF.

An "offset" will be calculated as "low limit of memory range" minus "Lowest Address". Each data record will then be loaded at the address specified in the record plus the "offset". Data will be loaded only within the "memory range" as determined above. NOTE: BASIC programs must be LOADed and SAVEd in BASIC, not in FCS.

READ

The READ command allows retrieval of information on any part of the diskette without regard to the directory or program boundaries. The command is of the form:

READ <Device Name:> Start Block Memory Spec

For example:

READ CD0: 20 7000-7FFF

reads 4096 bytes (1000 HEX) from the internal disk drive starting at block 32 (20 HEX) into the display memory at 7000-7FFF.

RENAME

The RENAME command allows the user, in one step, to change the file name, file, type and the version number separately or collectively without changing the information stored in the program. The statement is of the form:

RENAME <Device Name:> File Spec TO File Spec

For example:

REN TEST.PRG;1 TO NWTEST.PRG;2

renames the file TEST.PRG;1 to NWTEST.PRG;2.

RUN

The RUN command is used to load and execute machine-code programs. Only two file are permitted with the RUN command: .PRG and .LDA. The default file type is .PRG. To execute an .LDA file the .LDA extension must be specified. The RUN command is of the form:

RUN <Device Name:> File Spec

For example:

RUN CHESS

loads and executes a file CHESS.PRG from the default device.

SAVE

The SAVE command allows the user to save any type of data, program, or display in a file on a diskette. The command is of the form:

```
SAVE <Device Name:>  File Spec  Memory Spec  Start Address
        Actual Address
```

For example:

```
SAVE SCREEN.DSP 6000 1000
```

or

```
SAVE SCREEN.DSP 6000-6FFF
```

will save the screen display in a file called SCREEN.DSP.

WRITE

The WRITE command allows information to be written anywhere on the diskette without regard to the directory or previous program boundaries, and is of the form:

```
WRITE <Device Name:>  Start Block Number  Memory Spec
```

NOTE: It is possible to destroy the FCS directory information using the WRITE command. Care should always be taken when using this command.

APPENDICES

A. DISK BASIC

A.1 BASIC Statements

The following summary of BASIC statements defines the general format for each statement and gives a brief explanation. Optional items are enclosed in angle brackets, '<' and '>'. The following items in the syntax descriptions are used to represent different types of variables and expressions:

var	- numeric or string variable
nvar	- numeric variable
svar	- string varaible
expr	- numeric or string expression
nexpr	- numeric expression
sexpr	- string expression

STATEMENT SYNTAX AND DESCRIPTION

CLEAR — CLEAR <nexpr>
Clears all variables and optionally sets the string space size to nexpr bytes.

CONT — CONT
Continues execution after CTRL/J or ↓ (LINEFEED).

DATA — DATA value list
Defines data values to be read using the READ statement.

DEF — DEF FN nvar (nvar) = nexpr
Defines a user function to be used in the program.

DIM — DIM var(nexpr <,...,nexpr>) <,...>
Reserves space for lists and tables according to subscripts specified after variable name. Up to 255 dimensions.

END — END
Terminates program execution.

FILE "N" — FILE "N",filename,records,record size,blocking factor
Creates a new random file with the specified number of records (1-32767), record size (1-32767 bytes), and blocking factor (1-255). File name is a string expression containing a valid FCS file name.

FILE "R"

FILE "R",filenumber,filename,buffers <;records,record size, blocking factor>
Opens a random file with the specified file number (1-127) and number of buffers (1-255).

FILE "A"

FILE "A",file,current record <,records, record size, blocking factor>
Finds the attributes for the specified file.

FILE "C"

FILE "C",file1 <,...>
Closes the specified files and releases the buffer space.

FILE "D"

FILE "D",file1 <,...>
Writes any modified buffers for the specified files immediately to the corresponding devices.

FILE "T"

FILE "T" <,line number>
Causes file errors to trap to the specified line number. No line number turns the file error trapping off.

FILE "E"

FILE "E",file,error,line number
Finds the disk error number and location of the last file error.

FOR

FOR nvar = nexpr1 TO nexpr2 <STEP nexpr3>
Sets up a loop to be executed the specified number of times.

GET

GET file<,record<,first>>;nvar,svar[byte count],...
Reads from the record in the file starting from the first byte into the variables in the list. String variables must have a byte count (1-255).

GOSUB

GOSUB line number
Used to transfer control to the specified line number of a subroutine.

GOTO

GOTO line number
Used to unconditionally transfer control to the specified line number.

IF

IF nexpr GOTO line number
IF nexpr THEN line number
Used to conditionally transfer control to the specified line number.

IF nexpr THEN statement <:statement:...>
Used to conditionally execute BASIC statements

INPUT

INPUT <"string";> var <,var,...>
Used to input data from the terminal, prompts with either "?" or the optional quoted string as the prompt.

LIST

LIST <line number>
Prints the user program currently in memory on the CRT display, optionally, starting from the specified line number.

LOAD

LOAD filename
Loads the specified file. If no extension is specified, then a BASIC program is loaded; otherwise, the .ARY extension loads the specified numeric array, and the .DAT extension loads the specified data into memory after BASIC's workspace.

NEXT

NEXT <nvar <,nvar,...>>
Placed at the end of a FOR loop to return control to the FOR statement.

ON

ON nexpr GOSUB line number <,line number,...>
Multiple GOSUB statement. Transfers control to the line number specified by nexpr.

ON nexpr GOTO line number <,line number,...>
Multiple GOTO statement. Transfers control to the line number specified by nexpr.

OUT

OUT port,nexpr
Outputs the specified nexpr (0-255) to the 8080 port (0-255). CAUTION: Do not output to the CRT controller chips ports (96-111).

PLOT

PLOT nexpr <,nexpr,...>
Sends the one byte results (0-255) of the expressions to the CRT display.

POKE

POKE location,nexpr
Causes the one byte result of nexpr to be placed in the specified memory location (-32768 to 65535).

PRINT

PRINT expr <,expr,...>
PRINT expr <;expr;...>
Prints the results of the expressions in the list. Commas are used for normal spacing, and semicolons are used for compressed spacing. If either a comma or a semicolon is the last item in the print list, the carriage return is suppressed.

PRINT SPC(nexpr)
Prints the specified number of spaces. May be placed anywhere in the print list.

PRINT TAB(nexpr)
Tabs to the specified column. May be placed anywhere in the print list.

? Equivalent to the keyword PRINT.

PUT

PUT file <,record<,first>>; nexpr,sexpr[byte count] <,...>
Writes the expressions in the list to the record in the file starting from the first byte. String expressions must have a byte count.

READ

READ var <,var,...>
Used to assign the values in DATA statements to the variables specified in the list.

REM

REM comment
Used to insert explanatory comments in a BASIC program.

RESTORE

RESTORE <line number>
Resets the data pointer to either the first DATA statement or optionally to the specified line number.

RETURN

RETURN
Returns program control to the statement following the last executed GOSUB statement.

RUN

RUN <line number>
Executes the BASIC program in memory, optionally, starting at the specified line number.

SAVE

SAVE filename
Saves the specified file. If no extension is specified, the current BASIC program in memory is saved; otherwise, the .ARY extension saves the specified numeric array, and the .DAT extension saves the data in memory after BASIC's workspace.

WAIT

WAIT port, nexpr1 <,nexpr2>
Reads from the specified 8080 port and exclusive OR's the result with nexpr2 (0 if not present), and then AND's with nexpr1. The program waits until the result is zero before continuing.

:

statement : statement < : statement : ... >
A colon is used to separate statements in a multiple statement line.

A.2 BASIC Operators

SYMBOL	FUNCTION
=	Assignment or equality test (DISK BASIC does not allow the LET statement)
-	Negation or Subtraction
+	Addition or String Concatenation
*	Multiplication
/	Division
^	Exponentiation
NOT	Logical or One's complement (2 byte integer)
AND	Logical or Bitwise AND (2 byte integer)
OR	Logical or Bitwise OR (2 byte integer)
=,<,>,<=, =<,>=,=>, <>	Relational tests (result is TRUE = -1 or FALSE = 0)

The precedence of operators is:

1. Expressions in parentheses
2. Exponentiation (A^B)
3. Negation (-X)
4. *,/
5. +,-
6. Relational Operators (=,<>,<,>,<=,>=)
7. NOT
8. AND
9. OR

A.3 Standard Mathematical Functions

BASIC provides functions to perform certain standard mathematical operations such as square roots, logarithms, etc.

These functions have three or four letter call names followed by a parenthesized argument. They are predefined and may be used anywhere in a program.

CALL NAME	FUNCTION
ABS(x)	Returns the absolute value of x.
ATN(x)	Returns the arctangent of x as an angle in radians in range $+\pi/2$), where $\pi = 3.14159$.
CALL(x)	Call the user machine language program at decimal location 33282. (8202 HEX) D,E registers have value of X and D,E registers must have Y on return from machine language routine.
COS(x)	Returns the cosine of x radians.
EXP(x)	Returns the value of e^x where e = 2.71828.
FRE(x)	Returns number of free bytes not in use.
INT(x)	Returns the greatest integer less than or equal to x.
INP(x)	Returns a byte from input port x. The range for x is 0 to 255.
LOG(x)	Returns the natural logarithm of x.
PEEK(x)	Returns a byte from memory address $-32768<x<65535$; if x is negative the memory address is 65536+x.
POS(x)	Returns the value of the current cursor position between 0 and 63.
RND(x)	Returns a random number between 0 and 1.
SGN(x)	Returns a -1, 0, or 1, indicating the sign of x.
SIN(X)	Returns the sine of x radians.
SPC(x)	Causes x spaces to be generated. (Valid only in a PRINT statement).
SQR(x)	Returns the square root of x.
TAB(x)	Causes the cursor to space over to column number x. (Valid only in a PRINT statement).

TAN(x)	Returns the tangent of x radians.

The argument x to the functions can be a constant, a variable, an expression, or another function. Square brackets cannot be used as the enclosing characters for the argument x, e.g. SIN[x] is illegal.

Function calls, consisting of the function name followed by a parenthesized argument, can be used as expressions anywhere that expressions are legal.

Values produced by the functions SIN(x), COS(x), ATN(x), SQR(x), EXP(x), and LOG(x) have six significant digits.

A.4 Standard String Functions

Like the intrinsic mathematical functions (e.g., SIN, LOG), BASIC contains various functions for use with character strings. These functions allow the program to access parts of a string, determine the number of characters in a string, generate a character string corresponding to a given number or vice versa, and perform other useful operations. The various functions available are summarized in the following table.

CALL NAME	FUNCTION
ASC(x$)	Returns the eight bit internal ASCII code (0-255) for the one-character string. If the argument contains more than one character, then the code for the first character in the string is returned. A value of 0 is returned if the argument is a null string (LEN(x$) = 0). See ASCII codes in Appendix E.
CHR$(x)	Generates a one-character string having the ASCII value of x where x is a number in the range 0 to 255. Only one character can be generated.
FRE(x$)	Returns number of free string bytes. (See CLEAR statement in 3.11)
LEFT$(x$,I)	Returns left-most I characters of string (x$). If I>LEN(x$), then x$ is returned.
LEN(x$)	Returns the number of characters in the string x$, with non-printing characters and blanks being counted.
MID$(x$,I,J)	J is optional. Without J, returns right-most characters from x$ beginning with the Ith character. If I>LEN(x$), MID$ returns the null string. With 3 arguments, it returns a string of length J of characters from x$ beginning with the Ith character. If J is greater than the number if characters in x$ to the right of I, MID$ returns the rest of the string. Argument ranges: 0<I<=255, 0<=J<=255.

RIGHT$(x$,I)	Returns right-most I characters of string (x$). If I>LEN(x$), then x$ is returned.
STR$(x)	Returns the string which represents the numeric value of x as it would be printed by a PRINT statement.
VAL(x$)	Returns the number represented by the string x$. If the first character of x$ is not +, -, or a digit, then the value 0 is returned.

In the above example, x$ and y$ represent any legal string expressions, and I and J represent any legal arithmetic expressions.

A.5 BASIC Error Codes

After an error occurs, BASIC returns to command level and types READY. Variable values and the program text remain intact, but the program cannot be continued and all GOSUB and FOR context is lost.

When an error occurs in a statement executed in immediate mode, no line number is printed.

Format of error messages:

Stored BASIC statement	XX ERROR IN YYYY
Immediate mode statement	XX ERROR

In both of the above examples, "XX" is the error code. The "YYYY" is the line number in which the error occurred in the indirect statement.

The following are the possible error codes and their meanings:

ERROR	MEANING
BS	Bad Subscript. An attempt was made to reference a matrix element which is outside the dimension of the matrix. This error can occur if the wrong number of dimensions is used in a matrix reference. For instance, A (1,1,1)=Z when A has been dimensioned DIM A(2,2).
DD	Double Dimension. After a matrix was dimensioned, another dimension statement for the same matrix was encountered. This error often occurs if a matrix has been given the default dimension 10 because a statement like A(I)=3 is encountered and then later in the program a DIM A(100) is found.
CF	Call Function error. The parameter passed to a mathematical or string function was out of range. CF errors can occur due to:

1. a negative matrix subscript (A(-1)=0)
2. an unreasonably large matrix subscript (>32767)
3. LOG with a negative or zero argument
4. SQR with a negative argument

5. A^B with A negative and B not an integer
6. a CALL(x) before the address of the machine language subroutine has been patched in
7. calls to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, PLOT, TAB, SPC or ON...GOTO/GOSUB with an improper argument

ID Illegal Direct. You cannot use an INPUT or DEF statement in immediate mode.

NF NEXT without FOR. The variable in a NEXT statement corresponds to no previously mentioned FOR statement.

OD Out of Data. A READ statement was executed but all of the DATA statements in the program have already been read. The program tried to read too much data or an insufficient number of data values were included in the program.

OM Out of Memory. Program too large, too many variables, or too many FOR loops, too many GOSUB's, too complicated an expression, or any combination of the above.

OV Overflow. The result of a calculation was too large to be represented in BASIC's numeric format. If an underflow occurs, zero is given as the result and execution continues without any error message being printed.

SN Syntax error. Missing parenthesis in an expression, illegal character in a line, incorrect punctuation, etc.

RG RETURN without GOSUB. A RETURN statement was encountered without a previous GOSUB statement being executed.

US Undefined Statement. An attempt was made to GOTO, GOSUB, or THEN to a statement which does not exist.

/0 Division by Zero.

CN Continue error. Attempt to continue a program when none exists, an error occurred, or after a new line was typed into the program.

LS Long String. Attempt was made by use of the concatenation operator to create a string more than 255 characters long.

OS Out of String Space. Use the CLEAR X statement to allocate more string space or use smaller strings or fewer string variables.

SL SAVE/LOAD error. (From disk operation.) Other error messages may also appear from the File Control System. See Appendix B.2.

ST String Temporaries. A string expression was too complex. Break it into two or more shorter expressions.

TM	Type Mismatch. The left hand side of an assignment statement was a numeric variable and the right hand side was string, or vice versa, or, a function which expected a string argument was given a numeric one or vice versa.
UF	Undefined Function. Reference was made to a user defined function which was never defined.

A.6 BASIC Random File Error Codes

ERROR	NUMBER	MEANING
EV		No error vector. No file error trap line number has been set with a FILE "T" statement.
BF	2	Bad file name. Improper FCS file name.
NO	4	File not open. The specified file number is not open.
AO	6	File already open. The specified file number is already in use.
FS	8	File size error. The file being created with the FILE "N" statement is too large or the file parameters on the file being opened with the FILE "R" statement are improper.
RO	10	Record overflow. Too many data bytes were either read from or written to the current record.
EF	12	End of file. Tried to read or write past the end of the file.
CO	14	Cant't open file. The specified file does not exist on the specified device. (Possibly a diskette or hardware problem.)
CC	16	Can't close file. The specified file can not be closed. (Usually a diskette or hardware problem.)
RE	18	FCS READ error. (Usually a diskette or hardware problem.)
WE	20	FCS WRITE error. (Usually a diskette or hardware problem.)

B. FCS (File Control System)

B.1 FCS Commands

The File Control System is entered by pressing (ESC) then D from the keyboard, or PLOT 27,4 from BASIC. (Only the first three letters of the command need to be typed in.) If (ESC), D is from the keyboard then BASIC is terminated and must be re-entered by (ESC), E key sequence.

The following definitions will be used to describe the FCS commands:

( ) denotes mandatory element;
[ ] denotes optional element and if not specified, will result in the default type.

(Device name:) = [Device type] [Number] (:)
Device type is CD for COMPUCOLOR II Disk and number is either 0 or 1.
(Memory spec) = (Load address)(byte count) or (-end address)
All memory addresses are in HEX format.
(File Spec.) = (File name) [.Type] [;Version]
File name is any 6 characters. Type can be any three characters and PRG is the default type. Version is 0 to FF HEX. NOTE: After a default device type has been selected only the number of the device is required. The default device for the COMPUCOLOR II is CD0.

CAUTION: The COPY, DELETE, and DUPLICATE commands use the screen memory as a temporary buffer while performing the specified function.

COMMAND	SYNTAX AND DESCRIPTION
COPY	COPY [Device Name:] (File Spec) TO [Device Name:] [File Spec] Copies the specified file, usually, to another device. It uses the screen memory as a temporary buffer.
DELETE	DELETE [Device Name:] (File Spec) All File Spec options are required. Deletes the specified file, and repacks the disk and directory using the screen memory.
DEVICE	DEVICE [Device Name:] Sets and displays the current default Device Name.
DIRECTORY	DIRECTORY [Device Name:] Lists the directory for the default or specified device.

DUPLICATE	DUPLICATE (Device Name:) TO (Device Name:) Duplicates all the files on one diskette to another diskette on a second diskette using the screen memory as a temporary buffer.
EXIT "FCS"	ESC ESC or ESC E to return to BASIC. In BASIC, use PLOT 27,27 .
INITIALIZE	INITIALIZE (Device Name:) (Volume Name) [No. Dir. Blocks] Initializes the directory on the diskette currently in the specified device with the given Volume Name and number of directory blocks.
LOAD	LOAD [Device Name:] (File Spec) [Low Addr [Memory Spec]] Loads memory with a program. Defaults to .LDA type files written by the COMPUCOLOR II Assembler. (See Section 10.2 for complete details.)
READ	READ [Device Name:] (Start Block No.) (Memory Spec) Reads into memory from anywhere on the diskette starting at any block and ending where specified, without regard to program boundaries.
RENAME	RENAME [Device Name:] (File Spec) TO (File Spec) Allows any file to be renamed without changing any information in the file itself.
RUN	RUN [Device Name:] (File Spec) Loads and executes the specified program. The default type is .PRG.
SAVE	SAVE [Device Name:] (File Spec) (Memory Spec) [Start Address [Actual Address]] Saves memory image in the specified file. The Start Address and Actual Address default to the lower limit of the Memory Spec.
WRITE	WRITE [Device Name:] (Start Block No.) (Memory Spec) Writes memory image to the specified block on a diskette without regard to the FCS directory information and file boundaries. CAUTION: It is possible to destroy the FCS directory and file information on a diskette with this command.

B.2 FCS Error Codes

The numbers to the right of the code meanings refer to the list of error solutions that follows the code list.

MESSAGE	MEANING
EBLF	BAD LOAD FILE SPEC, 2
EBLK	INVALID BLOCK NUMBER, 2
ECOP	ERROR DURING COPY, 1 & 3
ECFB	CAN'T FIND BLOCK, 3
EDCS	DATA CRC ERROR, 3
EDEL	DELETE ERROR, 1 & 3
EDFN	DUPLICATE FILE NAME, 2
EDIR	DIRECTORY ERROR, 1 & 2
EDRF	DIRECTORY FULL, 4
EDSY	DATA SYNC CHARACTER ERROR, 1 & 3
EDUP	ERROR DURING DUPLICATE, 1 & 3
EFNF	FILE NOT FOUND, 2
EFRD	FILE READ ERROR, 3
EFWR	FILE WRITE ERROR, 3
EHCS	HEADER CRC ERROR, 3
EIVC	INVALID COMMAND, 2
EIVF	INVALID FUNCTION, 2
EIVD	INVALID DEVICE, 2
EIVP	INVALID PARAMETERS, 2
EIVU	INVALID UNIT, 2
EKBA	KEYBOARD ABORT, 4
EMDV	MISSING DEVICE NAME, 2
EMEM	MEMORY ERROR DURING READ, 4

EMFN	MISSING FILE NAME, 2
EMVN	MISSING VOLUME NAME, 2
EMVR	MISSING VERSION, 2
ENSA	NO START ADDRESS, 2
ENVE	NO VOLUME ENTRY IN DIRECTORY, 5
ERSZ	FILE TOO LARGE TO READ INTO ALLOCATED MEMORY, 2 & 4
ESIZ	DEVICE SIZES NOT SAME, 1
ESKF	SEEK FAILURE, 1
ESYN	SYNTAX ERROR, 2
EVFY	VERIFY FAILURE DURING WRITE, 3
EVOV	VERSION NUMBER OVERFLOW, 4
EWRF	WRITE FAILURE, 3
EWSF	FILE TOO LARGE TO WRITE ON DISKETTE, 2 & 4

Descriptions of Solutions to FCS Errors

1. Mechanical Problem--Jammed READ/WRITE head, loose disk drive, internal I/O connectors. Refer to COMPUCOLOR II Maintenance Manual.

2. Invalid User Input--Incorrect entry from user. Refer to FCS Commands, Section B.1.

3. Diskette Failure--Try a different diskette.

4. Error Message is self-explanatory.

5. Diskette Not Initialized--you need to initialize the diskette and possibly purchase a formatted COMPUCOLOR II blank diskette.

C. CRT COMMANDS

C.1 Control Codes

To enter a control code, hold down the CONTROL key while depressing the desired character key.

CONTROL CODE	CONTROL KEY	SECTION	EXPLANATION
0	@		NULL-Has no effect.
1	A	8.1.2	AUTO - Loads and runs a BASIC program named "MENU" from the disk drive.
2	B	9.5	PLOT - Enters graphic plot mode (see plot submodes), not allowed as a BASIC input character.
3	C	9.4	CURSOR X,Y - Enters X-Y cursor address mode for either visible cursor or blind cursor, used to go from BASIC to CRT MODE when typed as a BASIC input character.
4	D		Not used.
5	E		Not used.
6	F	9.2	CCI - The following character provides the 8 bit visible status word. Specifies Foreground, Background, Blink and Plot. (See Appendix C.2)
7	G		Not used.
8	H	9.4	HOME - Moves the cursor to top left corner of display.
9	I	9.4	TAB - Causes cursor to advance to next column--the tab columns are every 8 characters.
10	J	9.4	CURSOR DOWN or LINEFEED - Causes a break in BASIC execution of a program, causes the cursor to move down one line.
11	K	9.4	ERASE LINE - Causes the cursor to return to the beginning of the line and causes the complete line to be erased. Also causes the BASIC input line to be ignored.

12	L	9.4	ERASE PAGE - Causes the complete screen to be erased and the cursor to be moved to the home position. BASIC input ignores this character.
13	M	9.4	CARRIAGE RETURN - Causes the cursor to move to the beginning of the line it is presently on. Causes BASIC input to accept the typed line and process as a statement or input data.
14	N	9.2	A7 ON - Turns the A7 flag on. (2x character height and also stop bit.)
15	O	9.2	BLINK/A7 OFF - Turns the blink bit and A7 flag off.
16	P	9.2	BLACK KEY - Sets foreground color black if flag is off and background black if flag is on. (See codes 29 and 30 below.)
17	Q	9.2	RED KEY - Same as above with color red.
18	R	9.2	GREEN KEY - Same as above with color green.
19	S	9.2	YELLOW KEY - Same as above with color yellow.
20	T	9.2	BLUE KEY - Same as above with color blue.
21	U	9.2	MAGENTA KEY - Same as above with color magenta.
22	V	9.2	CYAN KEY - Same as above with color cyan.
23	W	9.2	WHITE KEY - Same as above with color white.
24	X	9.6	XMIT - Causes data to be transmitted from the visible cursor to the end of the page or until an FF,00 sequence is found in refresh RAM. Sends text characters with a linefeed and carriage return at end of each line. NOTE: Color status is not sent.
25	Y	9.4	CURSOR RIGHT - Causes the cursor to move right 1 position. On BASIC input displays previous character input.
26	Z	9.4	CURSOR LEFT - Causes the cursor to move left 1 position. On BASIC input deletes previous character from input buffer.

27	[		ESC - Provides an entry to the escape code table -- must be followed by one or more codes for proper operaton.
28	/	9.4	CURSOR UP - Causes the cursor to move up one line.
29	]	9.2	FG ON/FLAG OFF - Sets the flag bit off. If followed by one of the color keys it will set the foreground to that color. Also, does not change input codes in the range 96 to 127 that are to be stored in the display memory, i.e. the shifted alphabetic characters are displayed as shown in columns 6 and 7 in the COMPUCOLOR II character set in Appendix F. In plot mode OR's "ON" bits.
30	^	9.2	BG ON/FLAG ON - Sets the flag bit on. If followed by one of the color keys it will set the background to that color. With the FLAG on the shifted alphabetic characters 96 to 127 are converted into 0 to 31 when stored in the display memory, i.e. the characters displayed are shown in columns 0 and 1 in Appendix F. In plot mode XOR's "ON" bits.
31	_	9.2	BLINK ON - Turns on the blink bit which will blink the foreground color against the backround color.

C.2 STATUS WORD FORMAT

A7	A6	A5	A4	A3	A2	A1	A0
PLOT	BLINK	BACKGROUND COLOR			FOREGROUND COLOR		
		BLUE	GREEN	RED	BLUE	GREEN	RED

C.3 ESCAPE CODES

To enter an escape code sequence, depress the ESC key followed by the desired character key.

ESCAPE CODE	KEY	SECTION	EXPLANATION
0	@		Used for terminal control--not available for any other use.
1	A	9.4	Blind cursor mode.
2	B	9.3	Plot via color pad.
3	C	9.6	Transmit cursor X,Y position to RS-232C port.
4	D	10.1	Enters Disk File Control System (FCS) with CRT as output.
5	E	1.2	Re-entry to DISK BASIC.
6	F	9.7	Sets full duplex mode, not functional when in BASIC. The BREAK key restores to half duplex when in terminal mode.
7	G	10.1	Enters Disk File Control System (FCS) with RS-232C port as output.
8	H	9.7	Sets half duplex mode.
9	I	9.8	Causes a program jump to location 36864. (9000 HEX)
10	J	9.4	Sets write vertical mode.
11	K	9.4	Sets scroll up and write left to right mode.
12	L	9.7	Sets local mode.
13	M	9.6	Sends all output to the RS-232C port.
14	N		Set to ignore all inputs. In BASIC POKE must be used to reset back to normal, or hit CPU RESET.
15	O		Not used.
16	P		Not used.
17	Q		Not used.

18	R	9.6	Baud rate selection mode. A7 on = 1 stop bit, A7 off = 2 stop bits. See Appendix C.4 below for the next key to specify the baud rate.
19	S	9.8	Causes a program jump to location 40960. (A000 HEX)
20	T	9.8	Causes a program jump to location 33280. (8200 HEX)
21	U		Not used.
22	V		Not used.
23	W	1.2	Initializes and transfers control to DISK BASIC 8001.
24	X	9.4	Sets terminal to page mode and write left to right mode.
25	Y	9.8	Test mode -- fill page with next character.
26	Z		Not used.
27	[	9.4	Visible cursor mode. Also used to exit FCS.
28	/		Not used.
29	]		Not used.
30	^	9.8	User definable escape code. Causes a program jump to locaton 33275. (81BF HEX)
31	_		Transfer control to the CRT mode.

C.4 BAUD RATE SELECTION

Number Key	1	2	3	4	5	6.	7
Baud Rate	110	150	300	1200	2400	4800	9600

C.5 GRAPHIC PLOT SUBMODES

DISK BASIC PLOT or RS-232C CODE	PLOT SUBMODE	OPTIONAL FUNCTION KEYBOARD
255	Plot Mode Escape	F 15
254	Character Plot	F 14
253	X Point Plot	F 13
252	Y Point Plot	F 12
251	X-Y Incremental Point Plot	F 11
250	X0 of X Bar Graph	F 10
249	Y of X Bar Graph	F 9
248	X max of X Bar Graph	F 8
247	Incremental X Bar Graph	F 7
246	Y0 of Y Bar Graph	F 6
245	X of Y Bar Graph	F 5
244	Y max of Y Bar Graph	F 4
243	Incremental Y Bar Graph	F 3
242	X0 Vector Plot	F 2
241	Y0 Vector Plot	F 1
240	Incremental Vector Plot	F 0

For incremental plot submodes see the format of the incremental direction codes below.

C.6 INCREMENTAL DIRECTION CODES

ΔX1		ΔY1		ΔX2		ΔY2	
A7	A6	A5	A4	A3	A2	A1	A0
+	-	+	-	+	-	+	-
80	40	20	10	8	4	2	1

D. INTERNAL FEATURES

D.1 Key Memory Locations

24576 to 28671 = Screen refresh RAM (Fast) 6000-6FFF HEX
28672 to 32767 = Screen refresh RAM (Slow) 7000-7FFF HEX
32940 = Points to maximum RAM used by BASIC
32980 = Points to start of BASIC source
32982 = Points to end of source and start of variables
32984 = Points to end of variables and start of arrays
32986 = Points to end of arrays
33209 = 0 to 59 seconds of Real Time Clock
33210 = 0 to 59 minutes of Real Time Clock
33211 = 0 to 23 hours of Real Time Clock
33215 = User ESCAPE jump vector
33218 = User output FLAG jump vector
33221 = User input FLAG jump vector
33224 = User timer no 2 jump vector
33228 = External output port buffer
33247 = Keyboard FLAG.
33249 = FCS output FLAG
33251 = Input port FLAG
33265 = BASIC output FLAG
33272 = Output port FLAG
33273 = LIST output FLAG
33278 = Keyboard character
33279 = Keyboard character ready FLAG
33282 = Location of CALL(x) jump
33285 = BASIC output vector location
33289 = Number of characters on terminal output
33433 = Start of BASIC source code
65535 = Maximum Amount of RAM

D.2 PORT ASSIGNMENTS

PORT # HEX	I/O PORT ADDRESS
0 - F	TMS 5501
10 - 1F	TMS 5501 Duplicate Addresses
20 - 5F	Not Assigned
60 - 6F	SMC 5027
70 - 7F	SMC 5027 Duplicate Addresses
80 - FF	Not Assigned

PORT #	TMS 5501 I/O CHIP (See Appendix G.2)
HEX DEC	
0 - 0	Read Serial Data in from RS-232C interface
1 - 1	Read Parallel Data from keyboard and disk
2 - 2	Read Interrupt Address on TMS 5501
3 - 3	Read Status on TMS 5501
4 - 4	Issue Discrete Command
5 - 5	Set Baud Rate on Serial I/O
6 - 6	Transmit Serial Data to RS-232C interface
7 - 7	Transmit Parallel Data to keyboard and disk (also controls Disk R/W)
8 - 8	Load Interrupt Mask Register
9 - 9	Interval Timer #1
A - 10	Interval Timer #2
B - 11	Interval Timer #3
C - 12	Interval Timer #4
D - 13	Interval Timer #5
E - 14	No Function
F - 15	No Function

PORT #	SMC 5027 CRT CHIP (See Appendix G.3)
HEX DEC	
60 - 96	Load Register 0 - Don't Load
61 - 97	Load Register 1 - Don't Load
62 - 98	Load Register 2 - Don't Load
63 - 99	Load Register 3 - Don't Load
64 - 100	Load Register 4 - Don't Load
65 - 101	Load Register 5 - Don't Load
66 - 102	Load Register 6 - Roll Register#
67 - 103	Processor Load Command - Don't Use
68 - 104	Read Cursor X Register
69 - 105	Read Cursor Y Register
70 - 106	Issue Reset Command - Don't Issue
6B - 107	Scroll up 1 line
6C - 108	Load Cursor X Register
6D - 109	Load Cursor Y Register
6E - 110	Load Start Timing - Don't Load
6F - 111	Self Load Commad - Don't Use

WARNING: Do not output any values to the SMC 5027 CRT chip.

D.3 COMPUCOLOR Fifty Pin Bus

PIN	DESIGNATION	PIN	DESIGNATION
1	+12V	26	D2 BUS
2	$\overline{MR}$	27	A2
3	$\overline{MW}$	28	D3 BUS
4	$\overline{I/O\ W}$	29	A3
5	Φ2 (+12V)	30	D7 BUS
6	Φ2 TTL	31	A4
7	Φ1 (+12V)	32	D6 BUS
8	17.9712 MHz	33	D4 BUS
9	SYNC	34	D5 BUS
10	$\overline{RESET}$	35	A6
11	-5V	36	D0 8080
12	+5V	37	A7
13	GND	38	D1 8080
14	$\overline{I/O\ R}$	39	A8
15	A10	40	D2 8080
16	READY	41	A14
17	NO CONNECTION	42	D3 8080
18	NO CONNECTION	43	D4 8080
19	HOLD	44	A9
20	A5	45	A13
21	A11	46	D7 8080
22	D0 BUS	47	A12
23	A0	48	A15
24	D1 BUS	49	D5 8080
25	A1	50	D6 8080

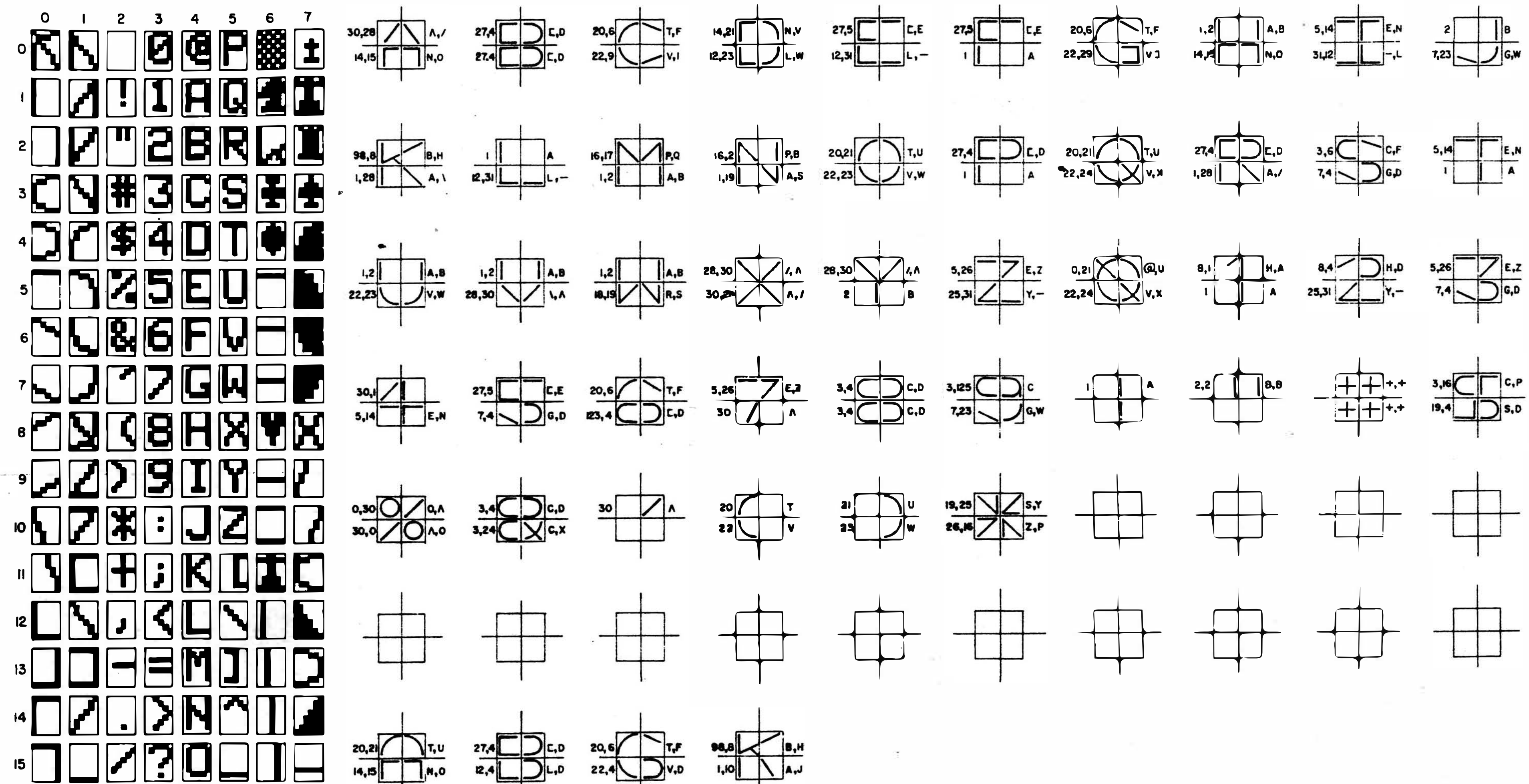
D.4 RS-232C INTERFACE

CPU EDGE CONNECTOR #	RS-232C PIN #	SIGNAL NAME AND LINE
1 PURPLE	1	AA Protective Ground
3 RED	2	BA Transmitted Data
5 BLUE	3	BB Received Data
7 WHITE	4	CA Request to Send
14 BLACK	7	AB Signal Ground
15 ORANGE	20	CD Data Terminal Ready

E. ASCII VALUES

<table>
<tr><th>DECIMAL</th><th>CHARACTER</th><th>DECIMAL</th><th>CHARACTER</th><th>DECIMAL</th><th>CHARACTER</th></tr>
<tr><td>000</td><td>NULL</td><td>048</td><td>0</td><td>096</td><td>`</td></tr>
<tr><td>001</td><td>AUTO</td><td>049</td><td>1</td><td>097</td><td>a</td></tr>
<tr><td>002</td><td>PLOT</td><td>050</td><td>2</td><td>098</td><td>b</td></tr>
<tr><td>003</td><td>CURSOR X,Y</td><td>051</td><td>3</td><td>099</td><td>c</td></tr>
<tr><td>004</td><td>(not used)</td><td>052</td><td>4</td><td>100</td><td>d</td></tr>
<tr><td>005</td><td>(not used)</td><td>053</td><td>5</td><td>101</td><td>e</td></tr>
<tr><td>006</td><td>CCI</td><td>054</td><td>6</td><td>102</td><td>f</td></tr>
<tr><td>007</td><td>(not used)</td><td>055</td><td>7</td><td>103</td><td>g</td></tr>
<tr><td>008</td><td>HOME</td><td>056</td><td>8</td><td>104</td><td>h</td></tr>
<tr><td>009</td><td>TAB</td><td>057</td><td>9</td><td>105</td><td>i</td></tr>
<tr><td>010</td><td>LINEFEED</td><td>058</td><td>:</td><td>106</td><td>j</td></tr>
<tr><td>011</td><td>ERASE LINE</td><td>059</td><td>;</td><td>107</td><td>k</td></tr>
<tr><td>012</td><td>ERASE PAGE</td><td>060</td><td><</td><td>108</td><td>l</td></tr>
<tr><td>013</td><td>RETURN</td><td>061</td><td>=</td><td>109</td><td>m</td></tr>
<tr><td>014</td><td>A7 ON</td><td>062</td><td>></td><td>110</td><td>n</td></tr>
<tr><td>015</td><td>BLINK/A7 OFF</td><td>063</td><td>?</td><td>111</td><td>o</td></tr>
<tr><td>016</td><td>BLACK KEY</td><td>064</td><td>@</td><td>112</td><td>p</td></tr>
<tr><td>017</td><td>RED KEY</td><td>065</td><td>A</td><td>113</td><td>q</td></tr>
<tr><td>018</td><td>GREEN KEY</td><td>066</td><td>B</td><td>114</td><td>r</td></tr>
<tr><td>019</td><td>YELLOW KEY</td><td>067</td><td>C</td><td>115</td><td>s</td></tr>
<tr><td>020</td><td>BLUE KEY</td><td>068</td><td>D</td><td>116</td><td>t</td></tr>
<tr><td>021</td><td>MAGENTA KEY</td><td>069</td><td>E</td><td>117</td><td>u</td></tr>
<tr><td>022</td><td>CYAN KEY</td><td>070</td><td>F</td><td>118</td><td>v</td></tr>
<tr><td>023</td><td>WHITE KEY</td><td>071</td><td>G</td><td>119</td><td>w</td></tr>
<tr><td>024</td><td>XMIT</td><td>072</td><td>H</td><td>120</td><td>x</td></tr>
<tr><td>025</td><td>CURSOR RIGHT</td><td>073</td><td>I</td><td>121</td><td>y</td></tr>
<tr><td>026</td><td>CURSOR LEFT</td><td>074</td><td>J</td><td>122</td><td>z</td></tr>
<tr><td>027</td><td>ESC</td><td>075</td><td>K</td><td>123</td><td>{</td></tr>
<tr><td>028</td><td>CURSOR UP</td><td>076</td><td>L</td><td>124</td><td>|</td></tr>
<tr><td>029</td><td>FG ON/FLAG OFF</td><td>077</td><td>M</td><td>125</td><td>}</td></tr>
<tr><td>030</td><td>BG ON/FLAG ON</td><td>078</td><td>N</td><td>126</td><td>~</td></tr>
<tr><td>031</td><td>BLINK ON</td><td>079</td><td>O</td><td>127</td><td>DEL</td></tr>
<tr><td>032</td><td>SPACE</td><td>080</td><td>P</td><td></td><td></td></tr>
<tr><td>033</td><td>!</td><td>081</td><td>Q</td><td></td><td></td></tr>
<tr><td>034</td><td>"</td><td>082</td><td>R</td><td></td><td></td></tr>
<tr><td>035</td><td>#</td><td>083</td><td>S</td><td></td><td></td></tr>
<tr><td>036</td><td>$</td><td>084</td><td>T</td><td></td><td></td></tr>
<tr><td>037</td><td>%</td><td>085</td><td>U</td><td></td><td></td></tr>
<tr><td>038</td><td>&</td><td>086</td><td>V</td><td></td><td></td></tr>
<tr><td>039</td><td>'</td><td>087</td><td>W</td><td></td><td></td></tr>
<tr><td>040</td><td>(</td><td>088</td><td>X</td><td></td><td></td></tr>
<tr><td>041</td><td>)</td><td>089</td><td>Y</td><td></td><td></td></tr>
<tr><td>042</td><td>*</td><td>090</td><td>Z</td><td></td><td></td></tr>
<tr><td>043</td><td>+</td><td>091</td><td>[</td><td></td><td></td></tr>
<tr><td>044</td><td>,</td><td>092</td><td>/</td><td></td><td></td></tr>
<tr><td>045</td><td>-</td><td>093</td><td>]</td><td></td><td></td></tr>
<tr><td>046</td><td>.</td><td>094</td><td>^</td><td></td><td></td></tr>
<tr><td>047</td><td>/</td><td>095</td><td>_</td><td></td><td></td></tr>
</table>

REVISIONS				
ZONE	LTR	DESCRIPTION	DATE	APPROVED
	1	REDRAWN	5-9-78	



ALTERNATE CHARACTER SHAPES

APPENDIX G.1

TMS 8080 MICROPROCESSOR

1. ARCHITECTURE

1.1 INTRODUCTION

The TMS 8080 is an 8-bit parallel central processing unit (CPU) fabricated on a single chip using a high-speed N-channel silicon-gate process. (See Figure 1). A complete microcomputer system with a 2-μs instruction cycle can be formed by interfacing this circuit with any appropriate memory. Separate 8-bit data and 16-bit address buses simplify the interface and allow direct addressing of 65,536 bytes of memory. Up to 256 input and 256 output ports are also provided with direct addressing. Control signals are brought directly out of the processor and all signals, excluding clocks, are TTL compatible.

1.2 THE STACK

The TMS 8080 incorporates a stack architecture in which a portion of external memory is used as a pushdown stack for storing data from working registers and internal machine status. A 16-bit stack pointer (SP) is provided to facilitate stack location in the memory and to allow almost unlimited interrupt handling capability. The CALL and RST (restart) instructions use the SP to store the program counter (PC) into the stack. The RET (return) instruction uses the SP to acquire the previous PC value. Additional instructions allow data from registers and flags to be saved in the stack.

1.3 REGISTERS

The TMS 8080 has three categories of registers: general registers, program control registers, and internal registers. The general registers and program control registers are listed in Table 1. The internal registers are not accessible by the programmer. They include the instruction register, which holds the present instruction, and several temporary storage registers to hold internal data or latch input and output addresses and data.

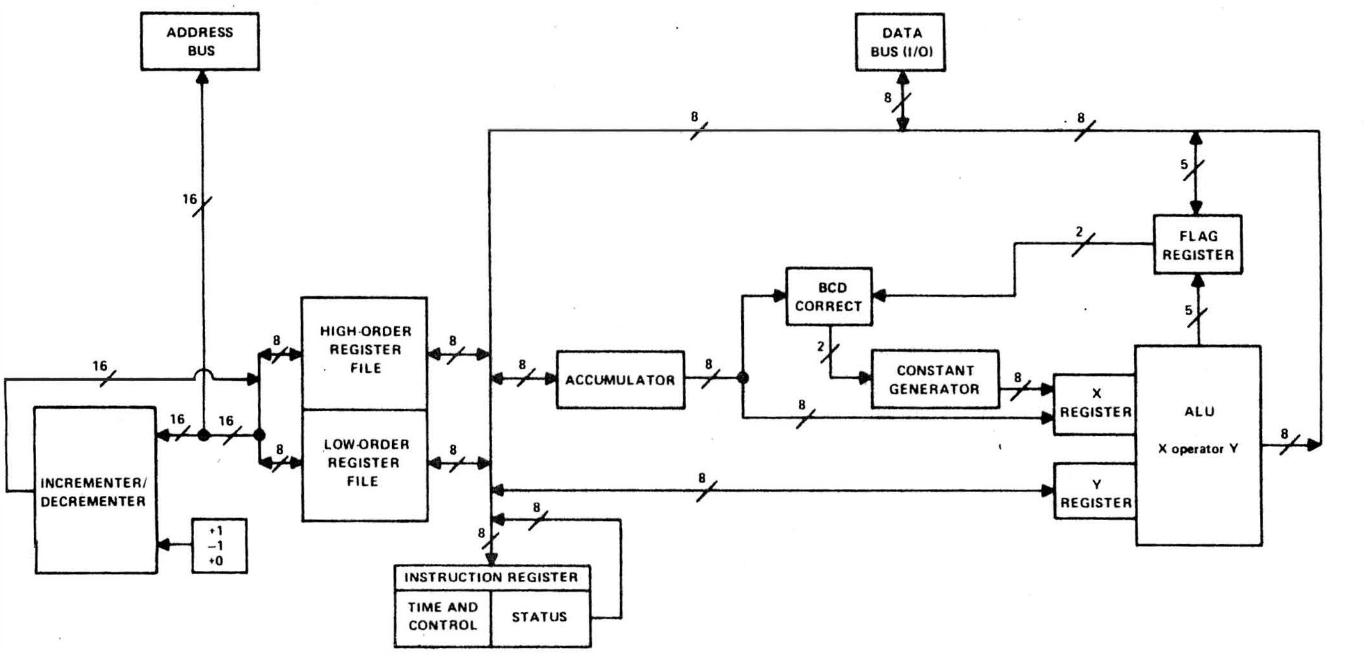


FIGURE 1—TMS 8080 FUNCTIONAL BLOCK DIAGRAM

1.4 THE ARITHMETIC UNIT

Arithmetic operations are performed in an 8-bit parallel arithmetic unit that has both binary and decimal capabilities. Four testable internal flag bits are provid d to facilitate program control, and a fifth flag is used for decimal corrections. Table 2 defines these flags and their operation. Decimal corrections are performed with the DAA instruction. The DAA corrects the result of binary arithmetic operation on BCD data as shown in Table 3.

1.5 STATUS AND CONTROL

Two types of status are provided by the TMS8080. Certain status is indicated by dedicated control lines. Additional status is transmitted on the data bus during the beginning of each instruction cycle (machine cycle). Table 4 indicates the pin functions of the TMS8080. Table 5 defines the status information that is presented during the beginning of each machine cycle (SYNC time) on the data bus.

1.6 I/O OPERATIONS

Input/output operations (I/O) are performed using the IN and OUT instructions. The second byte of these instructions indicates the device address (256 device addresses). When an IN instruction is executed, the input device address appears in duplicate on A7 through A0 and A15 through A8, along with $\overline{\text{WO}}$ and INP status on the data bus. The addressed input device then puts its input data on the data bus for entry into the accumulator. When an OUT instruction is executed, the same operation occurs except that the data bus has OUT status and then has output data.

Direct memory access channels (DMA) can be OR-tied directly with the data and address buses through the use of the HOLD and HLDA (hold acknowledge) controls. When a HOLD request is accepted by the CPU, HLDA goes high, the address and data lines are forced to a high-impedance or "floating" condition, and the CPU stops until the HOLD request is removed.

Interfacing with different speed memories is easily accomplished by use of the WAIT and READY pins. During each machine cycle, the CPU polls the READY input and enters a wait condition until the READY line becomes true. When the WAIT output pin is high, it indicates that the CPU has entered the wait state.

Designing interrupt driven systems is simplified through the use of vectored interrupts. At the end of each instruction, the CPU polls the INT input to determine if an interrupt request is being made. This action does not occur if the CPU is in the HOLD state or if interrupts are disabled. The INTE output indicates if the interrupt logic is enabled (INTE is high). When a request is honored, the INTA status bit becomes high, and an RST instruction may be inserted to force the CPU to jump to one of eight possible locations. Enabling or disabling interrupts is controlled by special instructions (EI or DI). The interrupt input is automatically disabled when an interrupt request is accepted or when a RESET signal is received.

1.7 INSTRUCTION TIMING

The execution time of the instructions varies depending on the operation required and the number of memory references needed. A machine cycle is defined to be a memory referencing operation and is either 3, 4, or 5 state times long. A state time (designated S) is a full cycle of clocks $\phi 1$ and $\phi 2$. (NOTE: The exception to this rule is the DAD instruction, which consists of 1 memory reference in 10 state times). The first machine cycle (designated M1) is either 4 or 5 state times long and is the "instruction fetch" cycle with the program counter appearing on the address bus. The CPU then continues with as many M cycles as necessary to complete the execution of the instruction (up to a maximum of 5). Thus the instruction execution time varies from 4 state times (several including ADDr) to 18 (XTHL). The WAIT or HOLD conditions may affect the execution time since they can be used to control the machine (for example to "single step") and the HALT instruction forces the CPU to stop until an interrupt is received. As the instruction execution is completed (or in the HALT state) the INT pin is polled for an interrupt. In the event of an interrupt, the PC will not be incremented during the next M1 and an RST instruction can be inserted.

TABLE 1
TMS 8080 REGISTERS

NAME	DESIGNATOR	LENGTH	PURPOSE
Accumulator	A	8	Used for arithmetic, logical, and I/O operations
B Register	B	8	General or most significant 8 bits of double register BC
C Register	C	8	General or least significant 8 bits of double register BC
D Register	D	8	General or most significant 8 bits of double register DE
E Register	E	8	General or least significant 8 bits of double register DE
H Register	H	8	General or most significant 8 bits of double register HL
L Register	L	8	General or least significant 8 bits of double register HL
Program Counter	PC	16	Contains address of next byte to be fetched
Stack Pointer	SP	16	Contains address of the last byte of data saved in the memory stack
Flag Register	F	5	Five flags (C, Z, S, P, C1)

NOTE: Registers B and C may be used together as a single 16-bit register, likewise, D and E, and H and L.

TABLE 2
FLAG DESCRIPTIONS

SYMBOL	TESTABLE	DESCRIPTION
C	YES	C is the carry/borrow out of the MSB (most significant bit) of the ALU (Arithment Logic Unit). A TRUE condition (C = 1) indicates overflow for addition or underflow for subtraction.
Z	YES	A TRUE condition (Z = 1) indicates that the output of the ALU is equal to zero.
S	YES	A TRUE condition (S = 1) indicates that the MSB of the ALU output is equal to a one (1).
P	YES	A TRUE condition (P = 1) indicates that the output of the ALU has even parity (the number of bits equal to one is even).
C1	NO	C1 is the carry out of the fourth bit of the ALU (TRUE condition). C1 is used only for BCD correction with the DAA instruction.

TABLE 3
FUNCTION OF THE DAA INSTRUCTION
Assume the accumulator(A) contains two BCD digits, X and Y

	7 … 4	3 … 0
ACC	X	Y

ACCUMULATOR BEFORE DAA				ACCUMULATOR AFTER DAA			
C	$A_7 \ldots A_4$	C1	$A_3 \ldots A_0$	C	$A_7 \ldots A_4$	C1	$A_3 \ldots A_0$
0	X < 10	0	Y < 10	0	X	0	Y
0	X < 10	1	Y < 10	0	X	0	Y + 6
0	X < 9	0	Y ≥ 10	0	X + 1	1	Y + 6
1	X < 10	0	Y < 10	1	X + 6	0	Y
1	X < 10	1	Y < 10	1	X + 6	0	Y + 6
1	X < 10	0	Y ≥ 10	1	X + 7	1	Y + 6
0	X ≥ 10	0	Y < 10	1	X + 6	0	Y
0	X ≥ 10	1	Y < 10	1	X + 6	0	Y + 6
0	X ≥ 9	0	Y ≥ 10	1	X + 7	1	Y + 6

NOTE: The corrections shown in Table 3 are sufficient for addition. For subtraction, the programmer must account for the borrow condition that can occur and give erroneous results. The most straight forward method is to set A = 99_{16} and carry = 1. Then add the minuend to A after subtracting the subtrahend from A.

TABLE 4
TMS 8080 PIN DEFINITIONS

SIGNATURE	PIN	I/O	DESCRIPTION
A15 (MSB)	36	OUT	A15 through A0 comprise the address bus. True memory or I/O device addresses appear on this 3-state bus during the first state time of each instruction cycle.
A14	39	OUT	
A13	38	OUT	
A12	37	OUT	
A11	40	OUT	
A10	1	OUT	
A9	35	OUT	
A8	34	OUT	
A7	33	OUT	
A6	32	OUT	
A5	31	OUT	
A4	30	OUT	
A3	29	OUT	
A2	27	OUT	
A1	26	OUT	
A0 (LSB)	25	OUT	
D7 (MSB)	6	IN/OUT	D7 through D0 comprise the bidirectional 3-state data bus. Memory, status, or I/O data is transferred on this bus.
D6	5	IN/OUT	
D5	4	IN/OUT	
D4	3	IN/OUT	
D3	7	IN/OUT	
D2	8	IN/OUT	
D1	9	IN/OUT	
D0 (LSB)	10	IN/OUT	
V_{SS}	2		Ground reference
V_{BB}	11		Supply voltage (−5 V nominal)
V_{CC}	20		Supply voltage (5 V nominal)
V_{DD}	28		Supply voltage (12 V nominal)
$\phi 1$	22	IN	Phase 1 clock.
$\phi 2$	15	IN	Phase 2 clock. See page 19 for $\phi 1$ and $\phi 2$ timing.
RESET	12	IN	Reset. When active (high) for a minimum of 3 clock cycles, the RESET input causes the TMS 8080 to be reset. PC is cleared, interrupts are disabled, and after RESET, instruction execution starts at memory location 0. To prevent a lockup condition, a HALT instruction must not be used in location 0.
HOLD	13	IN	Hold signal. When active (high) HOLD causes the TMS 8080 to enter a hold state and float (put the 3-state address and data bus in a high-impedance state). The chip acknowledges entering the hold state with the HLDA signal and will not accept interrupts until it leaves the hold state.
INT	14	IN	Interrupt request. When active (high) INT indicates to the TMS8080 that an interrupt is being requested. The TMS8080 polls INT during a HALT or at the end of an instruction. The request will be accepted except when INTE is low or the CPU is in the HOLD condition.
INTE	16	OUT	Interrupts enabled. INTE indicates that an interrupt will be accepted by the TMS 8080 unless it is in the hold state. INTE is set to a high logic level by the EI (Enable Interrupt) instruction and reset to a low logic level by the DI (Disable Interrupt) instruction. INTE is also reset when an interrupt is accepted and by a high on RESET.
DBIN	17	OUT	Data bus in. DBIN indicates whether the data bus is in an input or an output mode. (high = input, low = output).

TABLE 4 (CONTINUED)

SIGNATURE	PIN	I/O	DESCRIPTION
$\overline{WR}$	18	OUT	Write. When active (low) $\overline{WR}$ indicates a write operation on the data bus to memory or to an I/O port.
SYNC	19	OUT	Synchronizing control line. When active (high) SYNC indicates the beginning of each machine cycle of the TMS8080. Status information is also present on the data bus during SYNC for external latches.
HLDA	21	OUT	Hold acknowledge. When active (high) HLDA indicates that the TMS8080 is in a hold state.
READY	23	IN	Ready control line. An active (high) level indicates to the TMS 8080 that an external device has completed the transfer of data to or from the data bus. READY is used in conjunction with WAIT for different memory speeds.
WAIT	24	OUT	Wait status. When active (high) WAIT indicates that the TMS8080 has entered a wait state pending a READY signal from memory.

TABLE 5

TMS 8080 STATUS

SIGNATURE	DATA BUS BIT	DESCRIPTION
INTA	D0	Interrupt acknowledge.
$\overline{WO}$	D1	Indicates that current machine cycle will be a read (input) (high = read) or a write (output) (low = write) operation.
STACK	D2	Indicates that address is stack address from the SP.
HLTA	D3	HALT instruction acknowledge.
OUT	D4	Indicates that the address bus has an output device address and the data bus has output data.
M1	D5	Indicates instruction acquisition for first byte.
INP	D6	Indicates address bus has address of input device.
MEMR	D7	Indicates that data bus will be used for memory read data.

2. TMS 8080 INSTRUCTION SET

2.1 INSTRUCTION FORMATS

TMS 8080 instructions are either one, two, or three bytes long and are stored as binary integers in successive memory locations in the format shown below.

One-Byte Instructions

D7 D6 D5 D4 D3 D2 D1 D0	OP CODE

Two-Byte Instructions

D7 D6 D5 D4 D3 D2 D1 D0	OP CODE
D7 D8 D5 D4 D3 D2 D1 D0	OPERAND

Three-Byte Instructions

D7 D6 D5 D4 D3 D2 D1 D0	OP CODE
D7 D6 D5 D4 D3 D2 D1 D0	LOW ADDRESS OR OPERAND 1
D7 D6 D5 D4 D3 D2 D1 D0	HIGH ADDRESS OR OPERAND 2

2.2 INSTRUCTION SET DESCRIPTION

Operations resulting from the execution of TMS 8080 instructions are described in this section. The flags that are affected by each instruction are given after the description.

2.2.1 INSTRUCTION SYMBOLS

SYMBOL	DESCRIPTION	
$<b_2>$	Second byte of instruction	
$<b_3>$	Third byte of instruction	
r_a	Register #	Register Name
	000	B
	001	C
	010	D
	011	E
	100	H
	101	L
	111	A
r_b	Register #	Register Name
	00	BC
	01	DE
	10	HL
	11	SP
r_c	Register #	Register Name
	0	BC
	1	DE
r_d	Register #	Register Name
	00	BC
	01	DE
	10	HL
r_dL	Least significant 8 bits of r_d	
r_dH	Most significant 8 bits of r_d	
f	Flags	True condition
	Zero (Z)	Result is zero
	Carry (C)	Carry/borrow out of MSB is one
	Parity (P)	Parity of result is even
	Sign (S)	MSB of result is one
	Carry 1(C1)	Carry out of fourth bit is one
M	Memory address defined by registers H and L	
( )	Contents of specified address or register	
[ ]	Contents at address contained in specified register	
←	Is transferred to	
↔	Exchange	
Am	Bit m of A register (accumulator)	
{ }	Flags affected	
b_2	Single byte immediate operand	
b_3b_2	Double byte immediate operand	
$(nnn)_8$	(nnn) is an octal (base 8) number	

2.2.2 ACCUMULATOR GROUP INSTRUCTIONS

MNEMONIC	OPERANDS	BYTES	M CYCLES/ STATES	DESCRIPTION
ACI	b_2	2	2/7	(A) ← (A) + <b_2>+(carry), add the second byte of the instruction and the contents of the carry flag to register A and place in A. {C,Z,S,P,C1}
ADC	M	1	2/7	(A) ← (A) + (M) + (carry). {C,Z,S,P,C1}
ADC	r_a	1	1/4	(A) ← (A) + (r_a) + (carry). {C,Z,S,P,C1}
ADD	M	1	2/7	(A) ← (A) + (M), add the contents of M to register A and place in A. {C,Z,S,P,C1}
ADD	r_a	1	1/4	(A) ← (A) + (r_a). {C,Z,S,P,C1}
ADI	b_2	2	2/7	(A) ← (A) + <b_2>. {C,Z,S,P,C1}
ANA	M	1	2/7	(A) ← (A) AND (M), take the logical AND of M and register A and place in A. The carry flag will be reset low. {C,Z,S,P,C1}
ANA	r_a	1	1/4	(A) ← (A) AND (r_a). {C,Z,S,P,C1}
ANI	b_2	2	2/7	(A) ← (A) AND <b_2>. {C,Z,S,P,C1}
CMA		1	1/4	(A) ← ($\bar{A}$), complement A.
CMC		1	1/4	(carry) ← ($\overline{carry}$), complement the carry flag. {C}
CMP	M	1	2/7	(A) − (M), compare the contents of M to register A and set the flags accordingly. {C,Z,S,P,C1} (A) = (M) Z = 1 (A) ≠ (M) Z = 0 (A) < (M) C = 1 (A) (M) C = 0
CMP	r_a	1	1/4	(A) − (r_a). {C,Z,S,P,C1}
CPI	b_2	2	2/7	(A) − <b_2>. {C,Z,S,P,C1}
DAA		1	1/4	(A)← BCD correction of (A). The 8 bit A contents is corrected to form two 4 bit BCD digits after a binary arithmetic operation. A fifth flag C1 indicates the overflow from A_3. The carry flag C indicates the overflow from A_7 (See Table 3). {C,Z,S,P,C1}
DAD	r_b	1	1/10	(HL) ← (HL) + (r_b), add the contents of double register r_b to double register HL and place in HL. {C}
LDA	b_3b_2	3	4/13	(A)← [<b_3><b_2>]
LDAX	r_c	1	2/7	(A)← [(r_c)]
ORA	M	1	2/7	(A) ← (A) OR (M), take the logical OR of the contents of M and register A and place in A. The carry flag will be reset. {C,Z,S,P,C1}
ORA	r_a	1	1/4	(A) ← (A) OR (r_a). {C,Z,S,P,C1}
ORI	b_2	2	2/7	(A) ← (A) OR <b_2>. {C,Z,S,P,C1}
RAL		1	1/4	A_{m+1} ← A_m, A_0 ← (carry), (carry)←(A_7). Shift the contents of register A to the left one bit through the carry flag. {C}
RAR		1	1/4	A_m ← A_{m+1}, A_7←(carry), (carry)←A_0. {C}
RLC		1	1/4	A_{m+1} ← A_m, A_0←A_7 (carry)←(A_7). Shift the contents of register A to the left one bit. Shift A_7 into A and into the carry flag. {C}
RRC		1	1/4	A_m←A_{m+1}, A_7← A_0, (carry)←(A_0). {C}

MNEMONIC	OPERANDS	BYTES	M CYCLES/ STATES	DESCRIPTION
SBB	M	1	2/7	(A)←(A)−(M)−(carry), subtract the contents of M and the contents of the carry flag from register A and place in A. Two's complement subtraction is used and a true borrow causes the carry flag to be set (underflow condition). {C,Z,S,P,C1}
SBB	r_a	1	1/4	(A)←(A)−(r_a)−(carry). {C,Z,S,P,C1}
SBI	b_2	2	2/7	(A)←(A)−$<b_2>$−(carry). {C,Z,S,P,C1}
STA	b_3b_2	3	4/13	[$<b_3>$ $<b_2>$]←(A), store contents of A in memory address given in bytes 2 and 3.
STAX	r_c	1	2/7	[(r_c)]←(A), store contents of A in memory address given in BC or DE.
STC		1	1/4	(carry)←1, set carry flag to a 1 (true condition).
SUB	M	1	2/7	(A)←(A)−(M), subtract the contents of M from register A and place in A. Two's complement subtraction is used and a true borrow causes the carry flag to be set (underflow condition). {C,Z,S,P,C1}
SUB	r_a	1	1/4	(A)←(A)−(r_a). {C,Z,S,P,C1}
SUI	b_2	2	2/7	(A)←(A)−$<b_2>$. {C,Z,S,P,C1}
XRA	M	1	2/7	(A)←(A) XOR (M), take the exclusive OR of the contents of M and register A and place in A. The carry flag will be reset. {C,Z,S,P,C1}
XRA	r_a	1	1/4	(A)←(A) XOR (r_a). {C,Z,S,P,C1}
XRI	b_2	2	2/7	(A)←(A) XOR $<b_2>$. {C,Z,S,P,C1}

2.2.3 INPUT/OUTPUT INSTRUCTIONS

MNEMONIC	OPERANDS	BYTES	M CYCLES/ STATES	DESCRIPTION
IN	b_2	2	3/10	(A)←(input data from data bus), byte 2 is sent on bits A7-A0 and A15-A8 as the input device address. INP status is given on the data bus.
OUT	b_2	2	3/10	(Output data)←(A), byte 2 is sent on bits A7-A0 and A15-A8 as the output device address. OUT status is given on the data bus.

2.2.4 MACHINE INSTRUCTIONS

MNEMONIC	OPERANDS	BYTES	M CYCLES/ STATES	DESCRIPTION
HLT		1	2/7	Halt, all machine operations stop. All registers are maintained. Only an interrupt can return the TMS 8080 to the run mode. Note that a HLT should not be placed in location zero, otherwise after the reset pin is active, the TMS 8080 will enter a nonrecoverable state (until power is removed), i.e., in halt with interrupts disabled. This condition also occurs if a HLT is executed while interrupts are disabled. HLTA status is given on the data bus.
NOP		1	1/4	(PC)←(PC)+1, no operation.

2.2.5 PROGRAM COUNTER AND STACK CONTROL INSTRUCTIONS

MNEMONIC	OPERANDS	BYTES	M CYCLES/ STATES	DESCRIPTION
CALL	b_3b_2	3	5/17	[(SP)−1] [(SP)−2]←(PC), (SP)←(SP)−2, (PC)←$<b_3>$ $<b_2>$, transfer PC to the stack address given by SP, decrement SP twice, and jump unconditionally to address given in bytes 2 and 3.
Conditional call instructions for true flags:				
(f)			5/17 (Pass)	If (f) = 1, [(SP)−1] [(SP)−2]←(PC), (SP)←(SP)−2, (PS)←$<b_3>$ $<b_2>$, otherwise (PC)←(PC)+3. If the flag specified, f, is 1, then execute a call. Otherwise, execute the next instruction.
CC (carry)	b_3b_2	3	3/11 (Fail)	
CPE (parity)	b_3b_2	3		
CM (sign)	b_3b_2	3		
CZ (zero)	b_3b_2	3		
Conditional call instructions for false flags:				
(f)			5/17 (Pass)	If (f) = 0, [(SP)−1] [(SP)−2]←(PC), (SP)←(SP) −2, (PC)←$<b_3>$ $<b_2>$, otherwise (PC)←(PC)+3.
CNC (carry)	b_3b_2	3	3/11 (Fail)	
CPO (parity)	b_3b_2	3		
CP (sign)	b_3b_2	3		
CNZ (zero)	b_3b_2	3		
DI		1	1/4	Disable interrupts. INTE is driven false to indicate that no interrupts will be accepted.
EI		1	1/4	Enable interrupts. INTE is driven true to indicate that an interrupt will be accepted. Execution of this instruction is delayed to allow the next instruction to be executed before the INT input is polled.
JMP	b_3b_2	3	3/10	(PC)←$<b_3>$ $<b_2>$, jump unconditionally to address given in bytes 2 and 3.
Conditional jump instructions for true flags:				
(f)			3/10	If (f) = 1, (PC)←$<b_3><b_2>$, otherwise (PC)←(PC)+3. If the flag specified, f, is 1, execute a JMP. Otherwise, execute the next instruction.
JC (carry)	b_3b_2	3		
JPE (parity)	b_3b_2	3		
JM (sign)	b_3b_2	3		
JZ (zero)	b_3b_2	3		
Conditional jump instructions for false flags:				
(f)			3/10	If (f) = 0, (PC)←$<b_3>$ $<b_2>$, othewise (PC)←(PC)+3.
JNC (carry)	b_3b_2	3		
JPO (parity)	b_3b_2	3		
JM (sign)	b_3b_2	3		
JNZ (zero)	b_3b_2	3		
PCHL		1	1/5	(PC)←(HL)
POP	PSW	1	3/10	(F)←[(SP)], (A)←[(SP)+1], (SP)←(SP)+2, restore the last stack values addressed by SP into A and F. Increment SP twice.
POP	r_d	1	3/10	(r_{dL})←[(SP)], (r_{dH})←[(SP)+1], (SP)←(SP)+2.
PUSH	PSW	1	3/11	[(SP)−1]←(A), [(SP)−2]←(F), (SP)←(SP)−2, save the contents of A and F into the stack addressed by SP. Decrement SP twice.
PUSH	r_d	1	3/11	[(SP)−1]←(r_{dL}), [(SP)−2]←(r_{dH}), (SP)←(SP)−2.
RET		1	3/10	(PC)←[(SP)] [(SP)+1], (SP)←(SP)+2, return to program at memory address given by last values in the stack. The SP is incremented by two.

MNEMONIC	OPERANDS	BYTES	M CYCLES/ STATES	DESCRIPTION
Conditional return instructions for true flags:				
(f)			3/11 (Pass)	If $(f) = 1$, $(PC) \leftarrow [(SP)]$ $[(SP+1)]$, $(SP) \leftarrow (SP)+2$. If the flag specified, f, is 1, execute a RET. Otherwise, execute the next instruction.
RC (carry)	C	1	1/5 (Fail)	
RPE (parity)	P	1		
RM (sign)	S	1		
RZ (zero)	Z	1		
Conditional return instructions for false flags:				
(f)			3/11 (Pass)	If $(f) = 0$, $(PC) \leftarrow [(SP)]$ $[(SP)+1]$, $(SP) \leftarrow (SP)+2$.
RNC (carry)	C	1	1/5 (Fail)	
RPO (parity)	P	1		
RP (sign)	S	1		
RNZ (zero)	Z	1		
RST		1	3/11	$[(SP)-1]$ $[(SP)-2] \leftarrow (PC)$ $(SP) \leftarrow (SP)-2$, $(PC) \leftarrow 0000R0_8$ where R is a 3 bit field in RST ($RST=3R7_8$). Transfer PC to the stack address given by SP, decrement SP twice, and jump to the address specified by R.
SPHL		1	1/5	$(SP) \leftarrow (HL)$.

2.2.6 REGISTER GROUP INSTRUCTIONS

MNEMONIC	OPERANDS	BYTES	M CYCLES/ STATES	DESCRIPTION
DCR	M	1	3/10	$(M) \leftarrow (M)-1$, decrement the contents of memory location specified by H and L. {Z,S,P,C1}
DCR	r_a	1	1/5	$(r_a) \leftarrow (r_a)-1$, decrement the contents of register r_a. {Z,S,P,C1}
DCX	r_b	1	1/5	$(r_b) \leftarrow (r_b)-1$, decrement double registers BC, DE, HL, or SP.
INR	M	1	3/10	$(M) \leftarrow (M)+1$, increment the contents of memory location specified by H and L. {Z,S,P,C1 }
INR	r_a	1	1/5	$(r_a) \leftarrow (r_a)+1$, increment the contents of register r_a. {Z,S,P,C1}
INX	r_b	1	1/5	$(r_b) \leftarrow (r_b)+1$, increment double registers BC, DE, HL, or SP.
LHLD	b_3b_2	3	5/16	$(L) \leftarrow [\langle b_3\rangle \langle b_2\rangle]$; $(H) \leftarrow [\langle b_3\rangle \langle b_2\rangle+1]$, load registers H and L with contents of the two memory locations specified by bytes 3 and 2.
LXI	$r_bb_3b_2$	3	3/10	$(r_{bH}) \leftarrow \langle b_3\rangle$; $(r_{bL}) \leftarrow \langle b_2\rangle$, load double registers BC, DE, HL, or SP immediate with bytes 3, 2, respectively.
MVI	M,b_2	2	3/10	$(M) \leftarrow \langle b_2\rangle$, store immediate byte 2 in the address specified by HL
MVI	r_ab_2	2	2/7	$(r_a) \leftarrow \langle b_2\rangle$, load register r_a immediate with byte 2 of the instruction.
MOV	Mr_a	1	2/7	$(M) \leftarrow (r_a)$, store register r_a in the memory location addressed by H and L.
MOV	r_aM	1	2/7	$(r_a) \leftarrow (M)$, load register r_a with contents of memory addressed by HL.
MOV	$r_{a1}r_{a2}$	1	1/5	$(r_{a1}) \leftarrow (r_{a2})$, load register r_{a1} with contents of r_{a2}. r_{a2} contents remain unchanged.
SHLD	b_3b_2	3	5/16	$[\langle b_3\rangle \langle b_2\rangle] \leftarrow (L)$; $[\langle b_3\rangle \langle b_2\rangle+1)] \leftarrow (H)$, store the contents of H and L into two successive memory locations specified by bytes 3 and 2.
XCHG		1	1/4	$(H) \leftarrow (D)$; $(L) \leftarrow (E)$, exchange double registers HL and DE
XTHL		1	5/18	$(L) \leftarrow [(SP)]$, $(H) \leftarrow [(SP)+1]$, $(SP)=(SP)$, exchange the top of the stack with register HL.

2.3 INSTRUCTION SET OPCODES ALPHABETICALLY LISTED

MNEMONIC	BYTES	DESCRIPTION	REGISTER AFFECTED	POSITIVE-LOGIC HEX OPCODE D7–D4	POSITIVE-LOGIC HEX OPCODE D3–D0	CLOCK CYCLES*
ACI	2	Add immediate to A with carry†		C	E	7
ADC M	1	Add memory to A with carry†		8	E	7
ADC r	1	Add register to A with carry†	B	8	8	4
			C	8	9	
			D	8	A	
			E	8	B	
			H	8	C	
			L	8	D	
			A	8	F	
ADD M	1	Add memory to A†		8	6	7
ADD r	1	Add register to A†	B	8	0	4
			C	8	1	
			D	8	2	
			E	8	3	
			H	8	4	
			L	8	5	
			A	8	7	
ADI	2	Add immediate to A†		C	6	7
ANA M	1	AND memory with A†		A	6	7
ANAr	1	AND register with A†	B	A	0	4
			C	A	1	
			D	A	2	
			E	A	3	
			H	A	4	
			L	A	5	
			A	A	7	
ANI	2	AND immediate with A†		E	6	7
CALL	3	Call unconditional		C	D	17
CC	3	Call on carry		D	C	11/17
CM	3	Call on minus		F	C	11/17
CMA	1	Complement A		2	F	4
CMC	1	Complement carry‡		3	F	4
CMP M	1	Compare memory with A†		B	E	7
CMP r	1	Compare register with A				
			B	B	8	4
			C	B	9	
			D	B	A	
			E	B	B	
			H	B	C	
			L	B	D	
			A	B	F	
CNC	3	Call on no carry		D	4	11/17
CNZ	3	Call on no zero		C	4	11/17
CP	3	Call on positive		F	4	11/17
CPE	3	Call on parity even		E	C	11/17
CPI	2	Compare immediate with A†		F	E	7
CPO	3	Call on parity odd		E	4	11/17
CZ	3	Call on zero		C	C	11/17
DAA	1	Decimal adjust A†		2	7	4

*Two possible cycle times (11/17) indicate instruction cycles dependent on condition flags.
†All flags (C, Z, S, P, C1) affected.
‡Only carry flag affected.

MNEMONIC	BYTES	DESCRIPTION	REGISTER AFFECTED	POSITIVE-LOGIC HEX OPCODE D7–D4	D3–D0	CLOCK CYCLES
DAD B	1	Add B&C to H&L‡		0	9	10
DAD D	1	Add D&E to H&L‡		1	9	10
DAD H	1	Add H&L to H&L‡		2	9	10
DAD SP	1	Add stack pointer to H&L‡		3	9	10
DCR M	1	Decrement Memory§		3	5	10
DCR r	1	Decrement Register§	B	0	5	5
			C	0	D	
			D	1	5	
			E	1	D	
			H	2	5	
			L	2	D	
			A	3	D	
DCX B	1	Decrement B&C		0	B	5
DCX D	1	Decrement D&E		1	B	5
DCX H	1	Decrement H&L		2	B	5
DCX SP	1	Decrement stack pointer		3	B	5
DI	1	Disable interrupts		F	3	4
EI	1	Enable interrupts		F	B	4
HLT	1	Halt		7	6	7
IN	2	Input		D	B	10
INR M	1	Increment memory§		3	4	10
INR r	1	Increment register§	B	0	4	5
			C	0	C	
			D	1	4	
			E	1	C	
			H	2	4	
			L	2	C	
			A	3	C	
INX B	1	Increment B&C register		0	3	5
INX D	1	Increment D&E register		1	3	5
INX H	1	Increment H&L register		2	3	5
INX SP	1	Increment stack pointer		3	3	5
JC	3	Jump on carry		D	A	10
JM	3	Jump on minus		F	A	10
JMP	3	Jump unconditional		C	3	10
JNC	3	Jump on no carry		D	2	10
JNZ	3	Jump on no zero		C	2	10
JP	3	Jump on positive		F	2	10
JPE	3	Jump on parity even		E	A	10
JPO	3	Jump on parity odd		E	2	10
JZ	3	Jump on zero		C	A	10
LDA	1	Load A direct		3	A	13
LDAX B	1	Load A indirect		0	A	7
LDAX D	1	Load A indirect		1	A	7
LHLD	3	Load H&L direct		2	A	16
LXI B	3	Load immediate register pair B&C		0	1	10
LXI D	3	Load immediate register pair D&E		1	1	10
LXI H	3	Load immediate register		2	1	10
LXI SP	3	Load immediate stack pointer		3	1	10

‡Only carry flag affected.
§All flags except carry affected.

MNEMONIC	BYTES	DESCRIPTION	REGISTER AFFECTED	POSITIVE-LOGIC HEX OPCODE D7–D4	D3–D0	CLOCK CYCLES
MOV M,r	1	Move register to memory	B	7	0	7
			C	7	1	
			D	7	2	
			E	7	3	
			H	7	4	
			L	7	5	
			A	7	7	
MOV r,M	1	Move memory to register	B	4	6	7
			C	4	E	
			D	5	6	
			E	5	E	
			H	6	6	
			L	6	E	
			A	7	E	
MOV r_1, r_2	1	Move register to register	B,B	4	0	5
			B,C	4	1	
			B,D	4	2	
			B,E	4	3	
			B,H	4	4	
			B,L	4	5	
			B,A	4	7	
			C,B	4	8	
			C,C	4	9	
			C,D	4	A	
			C,E	4	B	
			C,H	4	C	
			C,L	4	D	
			C,A	4	F	
			D,B	5	0	
			D,C	5	1	
			D,D	5	2	
			D,E	5	3	
			D,H	5	4	
			H,L	5	5	
			D,A	5	7	
			E,B	5	8	
			E,C	5	9	
			E,D	5	A	
			E,E	5	B	
			E,H	5	C	
			E,L	5	D	
			E,A	5	F	
			H,B	6	0	
			H,C	6	1	
			H,D	6	2	
			H,E	6	3	
			H,H	6	4	
			H,L	6	5	
			H,A	6	7	
			L,B	6	8	

MNEMONIC	BYTES	DESCRIPTION	REGISTER AFFECTED	POSITIVE-LOGIC HEX OPCODE D7–D4	D3–D0	CLOCK CYCLES*
MOV r_1, r_2	1	Move register to register (continued)	L,C	6	9	
			L,D	6	A	
			L,E	6	B	
			L,H	6	C	
			L,L	6	D	
			L,A	6	F	
			A,B	7	8	
			A,C	7	9	
			A,D	7	A	
			A,E	7	B	
			A,H	7	C	
			A,L	7	D	
			A,A	7	F	
MVI M	2	Move immediate memory		3	6	10
MVI r	2	Move immediate register	B	0	6	7
			C	0	E	
			D	1	6	
			E	1	E	
			H	2	6	
			L	2	E	
			A	3	E	
NOP	1	No operation	4	0	0	4
ORA M	1	OR memory with A†		B	6	7
ORA r	1	OR register with A†	B	B	0	4
			C	B	1	
			D	B	2	
			E	B	3	
			H	B	4	
			L	B	5	
			A	B	7	
ORI	2	OR immediate with A†		F	6	7
OUT	2	Output		D	3	10
PCHL	1	H&L to program counter		E	9	5
POP B	1	Pop register pair B&C off stack		C	1	10
POP D	1	Pop register pair D&E off stack		D	1	10
POP H	1	Pop register pair H&L off stack		E	1	10
POP PSW	1	Pop A and flags off stack†		F	1	10
PUSH B	1	Push register pair B&C		C	5	11
PUSH D	1	Push register pair D&C		D	5	11
PUSH H	2	Push register pair H&L on stack		E	5	11
PUSH PSW	1	Push A and Flags on stack		F	5	11
RAL	1	Rotate A left through carry ‡		1	7	4
RAR	1	Rotate A right through carry‡		1	F	4
RC	1	Return on carry		D	8	5/11
RET	1	Return		C	9	10
RLC	1	Rotate A left‡		0	7	4
RM	1	Return on minus		F	8	5/11
RNC	1	Return on no carry		D	0	5/11
RNZ	1	Return on no zero		C	0	5/11
RP	1	Return on positive		F	0	5/11

*Two possible cycles times (11/17) indicate instruction cycles dependent on condition flags.
†All flags (C, Z, S, P, C1) affected.
‡Only carry flag affected.

MNEMONIC	BYTES	DESCRIPTION	REGISTER AFFECTED	POSITIVE-LOGIC HEX OPCODE D7–D4	D3–D0	CLOCK CYCLES*
RPE	1	Return on parity even		E	8	5/11
RPO	1	Return on parity odd		E	0	5/11
RRC	1	Rotate A right‡		0	F	4
RST	1	Restart				11
			$PC \leftarrow 0000_{16}$	C	7	
			$PC \leftarrow 0008_{16}$	C	F	
			$PC \leftarrow 0010_{16}$	D	7	
			$PC \leftarrow 0018_{16}$	D	F	
			$PC \leftarrow 0020_{16}$	E	7	
			$PC \leftarrow 0028_{16}$	E	F	
			$PC \leftarrow 0030_{16}$	F	7	
			$PC \leftarrow 0038_{16}$	F	F	
RZ	1	Return on Zero		C	8	5/11
SBB M	1	Subtract memory from A with borrow†		9	E	7
SBB r	1	Subtract register from A with borrow†	B	9	8	4
			C	9	9	
			D	9	A	
			E	9	B	
			H	9	C	
			L	9	D	
			A	9	F	
SBI	2	Subtract immediate from A with borrow†		D	E	7
SHLD	3	Store H&L direct		2	2	16
SPHL	1	H&L to stack pointer		F	9	5
STA	3	Store A direct		3	2	13
STAX B	1	Store A indirect		0	2	7
STAX D	1	Store A indirect		1	2	7
STC	1	Set carry‡		3	7	4
SUB M	1	Subtract memory from A†		9	6	7
SUB r	1	Subtract register from A†	B	9	0	4
			C	9	1	
			D	9	2	
			E	9	3	
			H	9	4	
			L	9	5	
			A	9	7	
SUI	2	Subtract immediate from A†		D	6	7
XCHG	1	Exchange D&E, H&L registers		E	B	4
XRA M	1	Exclusive OR memory with A†		A	E	7
XRA r	1	Exclusive OR register with A†	B	A	8	4
			C	A	9	
			D	A	A	
			E	A	B	
			H	A	C	
			L	A	D	
			A	A	F	
XRI	2	Exclusive OR immediate with A†		E	E	7
XTHL	1	Exchange top of stack H&L		E	3	18

*Two possible cycles times (11/17) indicate instruction cycles dependent on condition flags.
†All flags (C, Z, S, P, C1) affected.
‡Only carry flag affected.

APPENDIX G.2

TMS 5501 MULTIFUNCTION INPUT/OUTPUT CONTROLLER

1. INTRODUCTION

1.1 DESCRIPTION

The TMS 5501 is a multifunction input/output circuit for use with TI's TMS 8080 CPU. It is fabricated with the same N-channel silicon-gate process as the TMS 8080 and has compatible timing, signal levels, and power supply requirements. The TMS 5501 provides a TMS 8080 microprocessor system with an asynchronous communications interface, data I/O buffers, interrupt control logic, and interval timers.

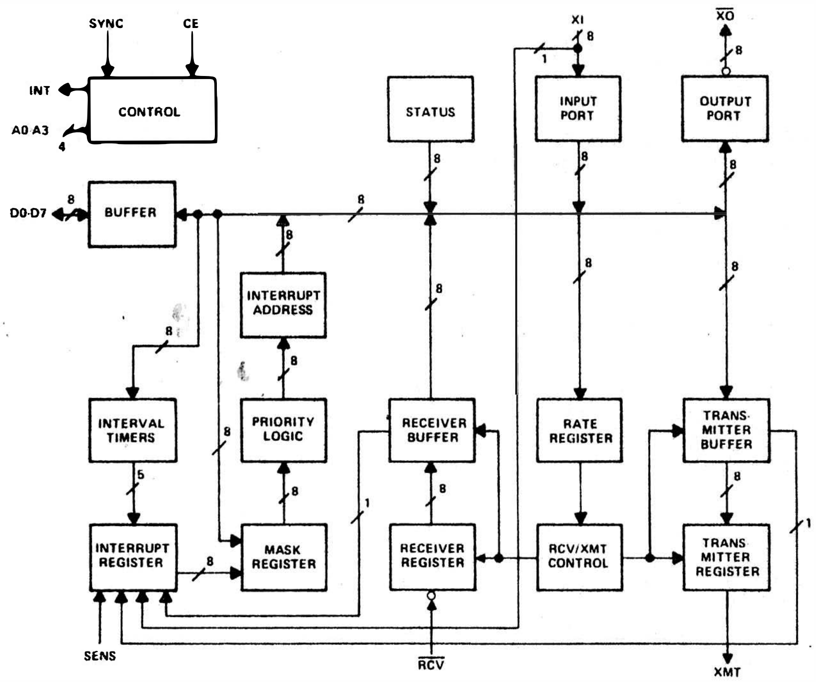


FIGURE 1–TMS 5501 BLOCK DIAGRAM

The I/O section of the TMS 5501 contains an eight-bit parallel input port and a separate eight-bit parallel output port with storage register. Five programmable interval timers provide time intervals from 64 μs to 16.32 ms.

The interrupt system allows the processor to effectively communicate with the interval timers, external signals, and the communications interface by providing TMS 8080-compatible interrupt logic with masking capability.

Data transfers between the TMS 5501 and the CPU are carried by the data bus and controlled by the interrupt, chip enable, sync, and address lines. The TMS 8080 uses four of its memory-address lines to select one of 14 commands to which the TMS 5501 will respond. These commands allow the CPU to:

---- read the receiver buffer
---- read the input port
---- read the interrupt address
---- read TMS 5501 status
---- issue discrete commands
---- load baud rate register
---- load the transmitter buffer
---- load the output port
---- load the mask register
---- load an interval timer

The commands are generated by executing memory referencing instructions such as MOV (register to memory) with the memory address being the TMS 5501 command. This provides a high degree of flexibility for I/O operations by letting the systems programmer use a variety of instructions.

1.2 SUMMARY OF OPERATION

Addressing the TMS 5501

A convenient method for addressing the TMS 5501 is to tie the chip enable input to the highest order address line of the CPU's 16-bit address bus and the four TMS 5501 address inputs to the four lowest order bits of the bus. This, of course, limits the system to 32,768 words of memory but in many applications the full 65,536 word memory addressing capability of the TMS 8080 is not required.

Communications Functions

The communications section of the TMS 5501 is an asynchronous transmitter and receiver for serial communications and provides the following functions:

Programmable baud rate — A CPU command selects a baud rate of 110, 150, 300, 1200, 2400, 4800, or 9600 baud.

Incoming character detection — The receiver detects the start and stop bits of an incoming character and places the character in the receive buffer.

Character transmission — The transmitter generates start and stop bits for a character received from the CPU and shifts it out.

Status and command signals — Via the data bus, the TMS 5501 signals the status of: framing error and overrun error flags; data in the receiver and transmitter buffers; start and data bit detectors; and end-of-transmission (break) signals from external equipment. It also issues break signals to external equipment.

Data Interface

The TMS 5501 moves data between the CPU and external devices through its internal data bus, input port, and output port. When data is present on the bus that is to be sent to an external device, a Load Output Port (LOP) command from the CPU puts the data on the $\overline{XO}$ pins of the TMS 5501 by latching it in the output port. The data remains in the port until another LOP command is received. When the CPU requires data that is present on the External Input (XI) lines, it issues a command that gates the data onto the internal data bus of the TMS 5501 and consequently onto the CPU's data bus at the correct time during the CPU cycles.

Interval Timers

To start a countdown by any of the five interval timers, the program selects the particular timer by an address to the TMS 5501 and loads the required interval into the timer via the data bus. Loading the timer activates it and it counts down in increments of 64 microseconds. The 8-bit counters provide intervals that vary in duration from 64 to 16,320 microseconds. Much longer intervals can be generated by cascading the timers through software. When a timer reaches zero, it generates an interrupt that typically will be used to point to a subroutine that performs a servicing function such as polling a peripheral or scanning a keyboard. Loading an interval value of zero causes an immediate interrupt. A new value loaded while the interval timer is counting overrides the previous value and the interval timer starts counting down the new interval. When an interval timer reaches zero it remains inactive until a new interval is loaded.

Servicing Interrupts

The TMS 5501 provides a TMS 8080 system with several interrupt control functions by receiving external interrupt signals, generating interrupt signals, masking out undersired interrupts, establishing the priority of interrupts, and generating RST instructions for the TMS 8080. An external interrupt is received on pin 22, SENS. An additional external interrupt can be received on pin 32, XI 7, if selected by a discrete command from the TMS 8080 (See Figure 4). The TMS 5501 generates an interrupt when any of the five interval timers count to zero. Interrupts are also generated when the receiver buffer is loaded and when the transmitter buffer is empty.

When an interrupt signal is received by the interrupt register from a particular source, a corresponding bit is set and gated to the mask register. A pattern will have previously been set in the mask register by a load-mask-register command from the TMS 8080. This pattern determines which interrupts will pass through to the priority logic. The priority logic allows an interrupt to generate an RST instruction to the TMS 8080 only if there is no higher priority interrupt that has not been accepted by the TMS 8080. The TMS 5501 prioritizes interrupts in the order shown below:

1st — Interval Timer #1
2nd — Interval Timer #2
3rd — External Sensor
4th — Interval Timer #3
5th — Receiver Buffer Loaded
6th — Transmitter Buffer Emptied
7th — Interval Timer #4
8th — Interval Timer #5 or an External Input (XI 7)

The highest priority interrupt passes through to the interrupt address logic, which generates the RST instruction to be read by the TMS 8080. See Table 3 for relationship of interrupt sources to RST instructions and Figures 6 and 8 for timing relationships.

The TMS 5501 provides two methods of servicing interrupts; an interrupt-driven system or a polled-interrupt system. In an interrupt-driven system, the INT signal of the TMS 5501 is tied to the INT input of the TMS 8080. The sequence of events will be: (1) The TMS 5501 receives (or generates) an interrupt signal and readies the appropriate RST instruction. (2) The TMS 5501 INT output, tied to the TMS 8080 INT input, goes high signaling the TMS 8080 that an interrupt has occured. (3) If the TMS 8080 is enabled to accept interrupts, it sets the INTA (interrupt acknowledge) status bit high at SYNC time of the next machine cycle. (4) If the TMS 5501 has previously received an interrupt-acknowledge-enable command from the CPU (see Bit 3, Paragraph 2.2.5), the RST instruction is transferred to the data bus.

In a polled-interrupt system, INT is not used and the sequence of events will be: (1) The TMS 5501 receives (or generates) an interrupt and readies the RST instruction. (2) The TMS 5501 interrupt-pending status bit (see Bit 5, Paragraph 2.2.4) is set high (the interrupt-pending status bit and the INT output go high simultaneously). (3) At the prescribed time, the TMS 8080 polls the TMS 5501 to see if an interrupt has occurred by issuing a read-TMS 5501-status command and reading the interrupt-pending bit. (4) If the bit is high, the TMS 8080 will then issue a read-interrupt-address command, which causes the TMS 5501 to transfer the RST instruction to the data bus as data for the instruction being executed by the TMS 8080.

1.3 APPLICATIONS

Communications Terminals

The functions of the TMS 5501 make it particularly useful in TMS 8080-based communications terminals and generally applicable in systems requiring periodic or random servicing of interrupts, generation of control signals to external devices, buffering of data, and transmission and reception of asynchronous serial data. As an example, a system configuration such as shown in Figure 2 can function as the controller for a terminal that governs employee entrance into a plant or security areas within a plant. Each terminal is identified by a central computer through ID switches. The central system supplies each terminal's RAM with up to 16 employee access categories applicable to that terminal. These categories are compared with an employee's badge character when he inserts his badge into the badge sensor. If a match is not found, a reject light will be activated. If a match is found, the terminal will transmit the employee's badge number and access category to the central system, and a door unlock solenoid will be activated for 4 seconds. The central computer then may take the transmitted information and record it along with time and date of access.

The TMS 4700 is a 1024 x 8 ROM that contains the system program, and the TMS 4036 is a 64 x 8 RAM that serves as the stack for the TMS 8080 and storage for the access category information. TTL circuits control chip-enable information carried by the address bus. Signals from the CPU gate the address bits from the ROM, the RAM, or the TMS 5501 onto the data bus at the correct time in the CPU cycle. The clock generator consists of four TTL circuits along with a crystal, needed to maintain accurate serial data assembly and disassembly with the central computer.

The TMS 5501 handles the asynchronous serial communication between the TMS 8080 and the central system and gates data from the badge reader onto the data bus. It also gates control and status data from the TMS 8080 to the door lock and badge reader and controls the time that the door lock remains open. The TMS 5501 signals the TMS 8080 when the badge reader or the communication lines need service. The functions that the TMS 5501 is to perform are selected by an address from the TMS 8080 with the highest order address line tied to the TMS 5501 chip enable input and the four lowest order lines tied to the address inputs.

2. OPERATIONAL AND FUNCTIONAL DESCRIPTION

This detailed description of the TMS 5501 consists of:

INTERFACE SIGNALS – a definition of each of the circuit's external connections

COMMANDS – the address required to select each of the TMS 5501 commands and a description of the response to the command.

2.1 INTERFACE SIGNALS

The TMS 5501 communicates with the TMS 8080 via four address lines: a chip enable line, an eight-bit bidirectional data bus, an interrupt line, and a sync line. It communicates with system components other than the CPU via eight external inputs, eight external outputs, a serial receiver input, a serial transmitter output, and an external sensor input. Table 1 defines the TMS 5501 pin assignments and describes the function of each pin.

TABLE 1
TMS 5501 PIN ASSIGNMENTS AND FUNCTIONS

SIGNATURE	PIN	DESCRIPTION
		INPUTS
CE	18	Chip enable–When CE is low, the TMS 5501 address decoding is inhibited, which prevents execution of any of the TMS 5501 commands.
A3	17	Address bus–A3 through A0 are the lines that are addressed by the TMS 8080 to select a particular TMS 5501 function.
A2	16	
A1	15	
A0	14	
SYNC	19	Synchronizing signal–The SYNC signal is issued by the TMS 8080 and indicates the beginning of a machine cycle and availability of machine status. When the SYNC signal is active (high), the TMS 5501 will monitor the data bus bits D0 (interrupt acknowledge) and D1 ($\overline{\text{WO}}$, data output function).
$\overline{\text{RCV}}$	5	Receiver serial data input line–$\overline{\text{RCV}}$ must be held in the inactive (high) state when not receiving data. A transition from high to low will activate the receive circuitry.

TABLE 1 (continued)
TMS 5501 PIN ASSIGNMENTS AND FUNCTIONS

SIGNATURE	PIN	DESCRIPTION
		INPUTS
XI 0	39	External inputs—These eight external inputs are gated to the data bus when the read-external-inputs function is addressed. External input n is gated to data bus bit n without conversion.
XI 1	38	
XI 2	37	
XI 3	36	
XI 4	35	
XI 5	34	
XI 6	33	
XI 7	32	
SENS	22	External interrupt sensing — A transition from low to high at SENS sets a bit in the interrupt register, which, if enabled, generates an interrupt to the TMS 8080.
		OUTPUTS
$\overline{XO}$ 0	24	External outputs—These eight external outputs are driven by the complement of the output register; i.e., if output register bit n is loaded with a high (low) from data bus bit n by a load-output register command, the external output n will be a low (high). The external outputs change only when a load-output-register function is addressed.
$\overline{XO}$ 1	25	
$\overline{XO}$ 2	26	
$\overline{XO}$ 3	27	
$\overline{XO}$ 4	28	
$\overline{XO}$ 5	29	
$\overline{XO}$ 6	30	
$\overline{XO}$ 7	31	
XMT	40	Transmitter serial data output line—This line remains high when the TMS 5501 is not transmitting.
		DATA BUS INPUT/OUTPUT
D0	13	Data bus — Data transfers between the TMS 5501 and the TMS 8080 are made via the 8-bit bidirectional data bus. D0 is the LSB. D7 is the MSB.
D1	12	
D2	11	
D3	10	
D4	9	
D5	8	
D6	7	
D7	6	
INT	23	Interrupt—When active (high), the INT output indicates that at least one of the interrupt conditions has occurred and that its corresponding mask-register bit is set.
		POWER AND CLOCKS
V_{SS}	4	Ground reference
V_{BB}	1	Supply voltage (−5 V nominal)
V_{CC}	2	Supply voltage (5 V nominal)
V_{DD}	3	Supply voltage (12 V nominal)
$\phi 1$	20	Phase 1 clock
$\phi 2$	21	Phase 2 clock

2.2 TMS 5501 COMMANDS

The TMS 5501 operates as input/output device for the TMS 8080. Functions are initiated via the TMS 8080 address bus and the TMS 5501 address inputs. Address decoding to determine the command function being issued is defined in Table 2.

TABLE 2
COMMAND ADDRESS DECODING
When Chip Enable Is High

#2 5501 PORT NO.	#1 5501 PORT NO.	A3	A2	A1	A0	COMMAND	FUNCTION	PARAGRAPH
16	0	L	L	L	L	Read receiver buffer	RBn → Dn	2.2.1
17	1	L	L	L	H	Read external inputs	XIn → Dn	2.2.2
18	2	L	L	H	L	Read interrupt address	RST → Dn	2.2.3
19	3	L	L	H	H	Read TMS 5501 status	(Status) → Dn	2.2.4
20	4	L	H	L	L	Issue discrete commands	See Fig.4	2.2.5
21	5	L	H	L	H	Load rate register	See Fig.4	2.2.6
22	6	L	H	H	L	Load transmitter buffer	Dn → TBn	2.2.7*
23	7	L	H	H	H	Load Output port	Dn → XOn	2.2.8
24	8	H	L	L	L	Load mask register	Dn → MRn	2.2.9
25	9	H	L	L	H	Load interval timer 1	Dn → Timer 1	2.2.10
26	10	H	L	H	L	Load interval timer 2	Dn → Timer 2	2.2.10
27	11	H	L	H	H	Load interval timer 3	Dn → Timer 3	2.2.10
28	12	H	H	L	L	Load interval timer 4	Dn → Timer 4	2.2.10
29	13	H	H	L	H	Load interval timer 5	Dn → Timer 5	2.2.10
30	14	H	H	H	L	No function		
31	15	H	H	H	H	No function		

* Important

RBn Receiver buffer bit n
Dn Data bus I/O terminal n
Xln External input terminal n
RST 11 $(1A_5)$ $(1A_4)$ $(1A_3)$ 1 1 1 (see Table 3)
TBn Transmit buffer bit n
XOn Output register bit n
MRn Mask register bit n

TABLE 3
RST INSTRUCTIONS

DATA BUS BIT 0	1	2	3	4	5	6	7	INTERRUPT CAUSED BY	TMS 5501
H	H	H	L	L	L	H	H	Interval Timer 1	Power Up
H	H	H	H	L	L	H	H	Interval Timer 2	User Timer
H	H	H	L	H	L	H	H	External Sensor	Real Time Clock
H	H	H	H	H	L	H	H	Interval Timer 3	Keyboard
H	H	H	L	L	H	H	H	Receiver Buffer	Rx RS-232
H	H	H	H	L	H	H	H	Transmitter Buffer	Tx RS-232
H	H	H	L	H	H	H	H	Interval Timer 4	Bell Timer
H	H	H	H	H	H	H	H	Not Available	CRT Executive Loop

The following paragraphs define the func.ions of the TMS 5501 commands.

2.2.1 Read receiver buffer

Addressing the read-receiver-buffer function causes the receiver buffer contents to be transferred to the TMS 8080 and clears the receiver-buffer-loaded flag.

2.2.2 Read external input lines

Addressing the read-external-inputs function transfers the states of the eight external input lines to the TMS 8080.

2.2.3 Read interrupt address

Addressing the read interrupt address function transfers the current highest priority interrupt address onto the data bus as read data. After the read operation is completed, the corresponding bit in the interrupt register is reset.

If the read-interrupt-address function is addressed when there is no interrupt pending, a false interrupt address will be read. TMS 5501 status function should be addressed in order to determine whether or not an interrupt condition is pending.

2.2.4 Read TMS 5501 status

Addressing the read-TMS 5501-status function gates the various status conditions of the TMS 5501 onto the data bus. The status conditions, available as indicated in Figure 3, are described in the following paragraphs.

BIT:	7	6	5	4	3	2	1	0
	START BIT DETECT	FULL BIT DETECT	INTRPT PENDING	XMIT BUFFER EMPTY	RCV BUFFER LOADED	SERIAL RCVD	OVERRUN ERROR	FRAME ERROR

FIGURE 3—DATA BUS ASSIGNMENTS FOR TMS 5501 STATUS

Bit 0, framing error

A high in bit 0 indicates that a framing error was detected on the last character received (either one or both stop bits were in error). The framing error flag is updated at the end of each character. Bit 0 of the TMS 5501 status will remain high until the next valid character is received.

Bit 1, overrun error

A high in bit 1 indicates that a new character was loaded into the receiver buffer before a previous character was read out. The overrun error flag is cleared each time the read-I/O-status function is addressed or a reset command is issued.

Bit 2, serial received data

Bit 2 monitors the receiver serial data input line. This line is provided as a status input for use in detecting a break and for test purposes. Bit 2 is normally high when no data is being received.

Bit 3, receiver buffer loaded

A high in bit 3 indiciates that the receiver buffer is loaded with a new character. The receiver-buffer-loaded flag remains high until the read-receiver-buffer function is addressed (at which time the flag is cleared). The reset function also clears this flag.

Bit 4, transmitter buffer empty

A high in bit 4 indicates that the transmitter buffer register is empty and ready to accept a character. Note, however, that the serial transmitter register may be in the process of shifting out a character. The reset function sets the transmitter-buffer-empty flag high.

Bit 5, interrupt pending

A high in bit 5 indicates that one or more of the interrupt conditions has occured and the corresponding interrupt is enabled. This bit is the status of the interrupt signal INT.

Bit 6, full bit detected

A high in bit 6 indicates that the first data bit of a receive-data character has been detected. This bit remains high until the entire character has been received or until a reset is issued and is provided for test purposes.

Bit 7, start bit detected

A high in bit 7 indicates that the start bit of an incoming data character has been detected. This bit remains high until the entire character has been received or until a reset is issued and is provided for test purposes.

2.2.5 Issue discrete commands

Addressing the discrete command function causes the TMS 5501 to interpret the data bus information according to the following descriptions. See Figure 4 for the discrete command format. Bits 1 through 5 are latched until a different discrete command is received.

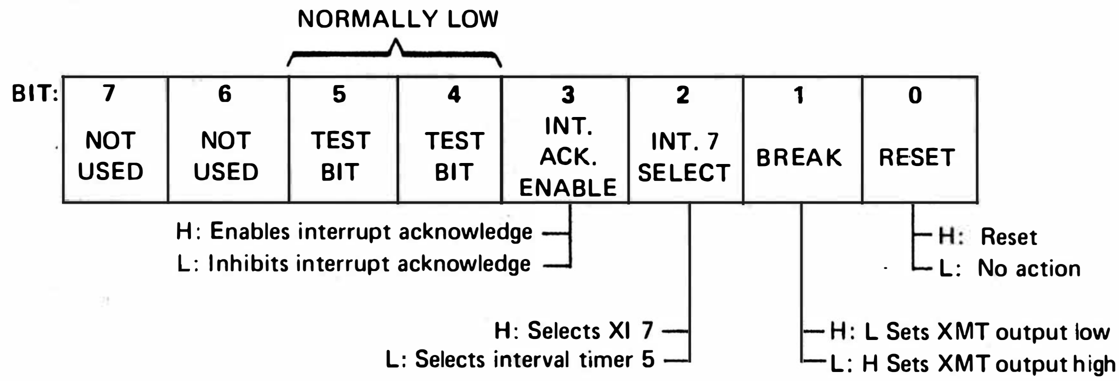


FIGURE 4—DISCRETE COMMAND FORMAT

Bit 0, reset

A high in bit 0 will cause the following:

1) The receiver buffer and register are cleared to the search mode including the receiver-buffer-loaded flag, the start-bit-detected flag, the full-bit-detected flag, and the overrun-error flag. The receiver buffer is not cleared and will contain the last character received.

2) The transmitter data output is set high (marking). The transmitter-buffer-empty flag is set high indicating that the transmitter buffer is ready to accept a character from the TMS 8080.

3) The interrupt register is cleared except for the bit corresponding to the transmitter buffer interrupt, which is set high.

4) The interval timers are inhibited.

A low in bit 0 causes no action. The reset function has no affect on the output port, the external inputs, interrupt acknowledge enable, the mask register, the rate register, the transmitter register, or the transmitter buffer.

Bit 1, break
A low in bit 1 causes the transmitter data output to be reset low (spacing).

If bit 0 and bit 1 are both high, the reset function will override.

Bit 2, interrupt 7 select
Interrupt 7 may be generated either by a low to high transition of external input 7 or by interval timer 5.

A high in bit 2 selects the interrupt 7 source to be the transition of external input 7. A low in bit 2 selects the interrupt 7 source to be interval timer 5.

Bit 3, interrupt acknowledge enable
The TMS 5501 decodes data bus (CPU status) bit 0 at SYNC of each machine cycle to determine if an interrupt acknowledge is being issued.

A high in bit 3 enables the TMS 5501 to accept the interrupt acknowledge decode. A low in bit 3 causes the TMS 5501 to ignore the interrupt acknowledge decode.

Bit 4 and bit 5 are used only during testing of the TMS 5501. For correct system operation both bits must be kept low.

Bit 6 and bit 7 are not used and can assume any value.

2.2.6 Load rate register

Addressing the load-rate-register function causes the TMS 5501 to load the rate register from the data bus and interpret the data bits (See Figure 5) as follows.

BIT:	7	6	5	4	3	2	1	0
	STOP BIT(s)	9600 baud	4800 baud	2400 baud	1200 baud	300 baud	150 baud	110 baud

Bit 7:
- H: One stop bit
- L: Two stop bits

FIGURE 5—DATA BUS ASSIGNMENTS FOR RATE COMMANDS.

Bits 0 through 6, rate select

The rate select bits (bits 0 through 6) are mutually exclusive, i.e., only one bit may be high. A high in bits 0 through 6 will select the baud rate for both the transmitter and receiver circuitry as defined below and in Figure 5:

Bit 0	110 baud
Bit 1	150 baud
Bit 2	300 baud
Bit 3	1200 baud
Bit 4	2400 baud
Bit 5	4800 baud
Bit 6	9600 baud

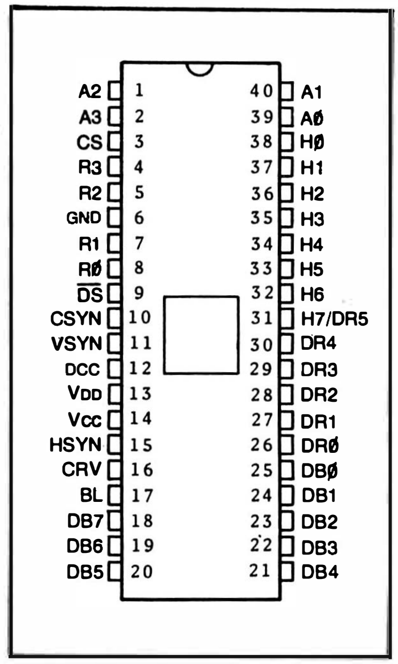
If more than one bit is high, the highest rate indicated will result. If bits 0 through 6 are all low, both the receiver and the transmitter circuitry will be inhibited.

CRT 5027
μPC FAMILY

APPENDIX G.3

CRT Video Timer-Controller VTAC

PIN CONFIGURATION



FEATURES

- ☐ Fully Programmable Display Format
  - Characters per data row (1-200)
  - Data rows per frame (1-64)
  - Raster scans per data row (1-16)
- ☐ Programmable Monitor Sync Format
  - Raster Scans/Frame (256-1023)
  - "Front Porch"
  - Sync Width
  - "Back Porch"
  - Interlace/Non-Interlace
  - Vertical Blanking
- ☐ Direct Outputs to CRT Monitor
  - Horizontal Sync
  - Vertical Sync
  - Composite Sync
  - Blanking
  - Cursor coincidence
- ☐ Programmed via:
  - Processor data bus
  - External PROM
  - Mask Option ROM
- ☐ Standard or Non-Standard CRT Monitor Compatible
- ☐ Refresh Rate: 60Hz, 50Hz,...
- ☐ Scrolling
  - Single Line
  - Multi-Line
- ☐ Cursor Position Registers
- ☐ Character Format: 5x7, 7x9,...
- ☐ Programmable Vertical Data Positioning
- ☐ Graphics Compatible
- ☐ Split-Screen Applications
  - Horizontal
  - Vertical
  - Programmable Wipes
- ☐ External Video Sync-Lock
- ☐ TTL Compatibility
- ☐ BUS Oriented
- ☐ High Speed Operation
- ☐ COPLAMOS® N-Channel Silicon Gate Technology

General Description

The CRT Video Timer-Controller Chip (VTAC) is a user programmable 40-pin COPLAMOS® n channel MOS/LSI device containing the logic functions required to generate all the timing signals for the presentation and formatting of interlaced and non-interlaced video data on a standard or non-standard CRT monitor.

With the exception of the dot counter, which may be clocked at a video frequency above 25 MHz and therefore not recommended for MOS implementation, all frame formatting, such as horizontal, vertical, and composite sync, characters per data row, data rows per frame, and raster scans per data row and per frame are totally user programmable. The data row counter has been designed to facilitate scrolling.

Programming is effected by loading seven 8 bit control registers directly off an 8 bit bidirectional data bus. Four register address lines and a chip select line provide complete microprocessor compatibility for program controlled set up. The device can be "self loaded" via an external PROM tied on the data bus as described in the OPERATION section. Formatting can also be programmed by a single mask option.

In addition to the seven control registers two additional registers are provided to store the cursor character and data row addresses for generation of the cursor video signal. The contents of these two registers can also be read out onto the bus for update by the program.

Bit 7, stop bits

Bit 7 determines whether one or two stop bits are to be used by both the transmitter and receiver circuitry. A high in bit 7 selects one stop bit. A low in bit 7 selects two stop bits.

2.2.7 Load transmitter buffer

Addressing the load-transmitter-buffer function transfers the state of the data bus into the transmitter buffer.

2.2.8 Load output port

Addressing the load-output-port function transfers the state of the data bus into the output port. The data is latched and remains on $\overline{XO\ 0}$ through $\overline{XO\ 7}$ as the complement of the data bus until new data is loaded.

2.2.9 Load mask register

Addressing the load-mask-register function loads the contents of the data bus into the mask register. A high in data bus bit n enables interrupt n. A low inhibits the corresponding interrupt.

2.2.10 Load timer n

Addressing the load-timer-n function loads the contents of the data bus into the appropriate interval timer. Time intervals of from 64 μs (data bus = LLLLLLLH) to 16,320 μs (data bus HHHHHHHH) are counted in 64-μs, steps. When the count of interval timer n reaches 0, the bit in the interrupt register that corresponds to timer n is set and an interrupt is generated. Loading all lows causes an interrupt immediately.

3. TMS 5501 ELECTRICAL AND MECHANICAL SPECIFICATIONS

3.1 ABSOLUTE MAXIMUM RATINGS OVER OPERATING FREE-AIR TEMPERATURE RANGE (UNLESS OTHERWISE NOTED)*

Supply voltage, V_{CC} (see Note 1)	−0.3 V to 20 V
Supply voltage, V_{DD} (see Note 1	−0.3 V to 20 V
Supply voltage, V_{SS} (see Note 1)	−0.3 V to 20 V
All input and output voltages (see Note 1)	−0.3 V to 20 V
Continuous power dissipation	1.1 W
Operating free-air temperature range	0°C to 70°C
Storage temperature range	−65°C to 150°C

*Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions beyond those indicated in the "Recommended Operating Conditions" section of this specification is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

NOTE 1: Under absolute maximum ratings voltage values are with respect to the normally most negative supply voltage, V_{BB} (substrate). Throughout the remainder of this data sheet, voltage values are with respect to V_{SS} unless otherwise noted.

3.2 RECOMMENDED OPERATING CONDITIONS

	MIN	NOM	MAX	UNIT
Supply voltage, V_{BB}	−4.75	−5	−5.25	V
Supply voltage, V_{CC}	4.75	5	5.25	V
Supply voltage, V_{DD}	11.4	12	12.6	V
Supply voltage, V_{SS}		0		V
High-level input voltage, V_{IH} (all inputs except clocks)	3.3		V_{CC}+1	V
High-level clock input voltage, $V_{IH(\phi)}$	V_{DD}−1		V_{DD}+1	V
Low-level input voltage, V_{IL} (all inputs except clocks) (see Note 2)	−1		0.8	V
Low-level clock input voltage, $V_{IL(\phi)}$ (see Note 2)	−1		0.6	V
Operating free-air temperature, T_A	0		70	°C

NOTE 2: The algebraic convention where the most negative limit is designated as minimum is used in this specification for logic voltage levels only.

MAXIMUM GUARANTEED RATINGS*

Operating Temperature Range .... 0°C to + 70°C
Storage Temperature Range .... −55°C to +150°C
Lead Temperature (soldering, 10 sec.) .... +325°C
Positive Voltage on any Pin, with respect to ground .... +18.0V
Negative Voltage on any Pin, with respect to ground .... −0.3V

*Stresses above those listed may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or at any other condition above those indicated in the operational sections of this specification is not implied.

NOTE: When powering this device from laboratory or system power supplies, it is important that the Absolute Maximum Ratings not be exceeded or device failure can result. Some power supplies exhibit voltage spikes or "glitches" on their outputs when the AC power is switched on and off. In addition, voltage transients on the AC power line may appear on the DC output. For example, the bench power supply programmed to deliver +12 volts may have large voltage transients when the AC power is switched on and off. If this possibility exists it is suggested that a clamp circuit be used.

ELECTRICAL CHARACTERISTICS (T_A=0°C to 70°C, V_{CC}=+5V±5%, V_{DD}=+12V±5%, unless otherwise noted)

Parameter	Min.	Typ.	Max.	Unit	Comments
D.C. CHARACTERISTICS					
INPUT VOLTAGE LEVELS					
Low Level, V_{IL}			0.8	V	
High Level, V_{IH}	V_{CC}−1.5		V_{CC}	V	
OUTPUT VOLTAGE LEVELS					
Low Level—V_{OL} for RØ-3			0.4	V	I_{OL}=3.2ma
Low Level—V_{OL} all others			0.4	V	I_{OL}=1.6ma
High Level—V_{OH} for RØ-3	2.4				I_{OH}=80μa
High Level—V_{OH} all others	2.4				I_{OH}=40μa
INPUT CURRENT					
Low Level, I_{IL}					
High Level, I_{IH}					
INPUT CAPACITANCE					
Data Bus, C_{IN}		10		pf	
Clock, C_{IN}		25		pf	
All other, C_{IN}		10		pf	
DATA BUS LEAKAGE in INPUT MODE					
I_{DB}					
I_{DB}					
POWER SUPPLY CURRENT					
I_{CC}		80		ma	
I_{DD}		40		ma	
A.C. CHARACTERISTICS					T_A= 25°C
DOT COUNTER CARRY					
frequency	0.2	4.0		MHz	Figure 1
PW_H	35			ns	Figure 1
PW_L	190			ns	Figure 1
t_r, t_f		10		ns	Figure 1
DATA STROBE					
$PW_{\overline{DS}}$		150		ns	Figure 2
ADDRESS, CHIP SELECT					
Set-up time		100		ns	Figure 2
Hold time		50		ns	Figure 2
DATA BUS—LOADING					
Set-up time		100		ns	Figure 2
Hold time		75		ns	Figure 2
DATA BUS—READING					
T_{DEL2}		100		ns	Figure 2, CL=50pf
OUTPUTS: HØ-7, HS, VS, BL, CRV, CS-T_{DEL1}		100		ns	Figure 1, CL=20pf
OUTPUTS: RØ-3, DRØ-5					
T_{DEL3}		1.0		μs	Figure 3, CL=20pf

Restrictions

1. Only one pin is available for strobing data into the device via the data bus. The cursor X and Y coordinates are therefore loaded into the chip by presenting one set of addresses and outputed by presenting a different set of addresses. Therefore the standard WRITE and READ control signals from most microprocessors must be "NORed" externally to present a single strobe ($\overline{DS}$) signal to the device.
2. An even number of scan lines per character row must be programmed in interlace mode. This is again due to pin count limitations which require that the least significant bit of the scan counter serve as the odd/even field indicator.
3. In interlaced mode the total number of character slots assigned to the horizontal scan must be even to insure that vertical sync occurs precisely between horizontal sync pulses.

Operation

The design philosophy employed was to allow the device to interface effectively with either a microprocessor based or hardwire logic system. The device is programmed by the user in one of two ways; via the processor data bus as part of the system initialization routine, or during power up via a PROM tied on the data bus and addressed directly by the Row Select outputs of the chip. (See figure 4). Seven 8 bit words are required to fully program the chip. Bit assignments for these words are shown in Table 1. The information contained in these seven words consists of the following:

Horizontal Formatting:

Characters/Data Row	A 3 bit code providing 8 mask programmable character lengths from 20 to 132. The standard device will be masked for the following character lengths; 20, 32, 40, 64, 72, 80, 96, and 132.
Horizontal Sync Delay	3 bits assigned providing up to 8 character times for generation of "front porch".
Horizontal Sync Width	4 bits assigned providing up to 16 character times for generation of horizontal sync width.
Horizontal Line Count	8 bits assigned providing up to 256 character times for total horizontal formatting.
Skew Bits	A 2 bit code providing from a 0 to 2 character skew (delay) between the horizontal address counter and the horizontal blank and sync signals to allow for retiming of video data prior to generation of composite video signal. The Cursor Video signal is also skewed as a function of this code.

Vertical Formatting:

Interlaced/Non-interlaced	This bit provides for data presentation with odd/even field formatting for interlaced systems. It modifies the vertical timing counters as described below. A logic 1 establishes the interlace mode.
Scans/Frame	8 bits assigned, defined according to the following equations: Let X = value of 8 assigned bits. 1) in interlaced mode—scans/frame = 2X + 513. Therefore for 525 scans, program X = 6 (00000110). Vertical sync will occur precisely every 262.5 scans, thereby producing two interlaced fields. Range = 513 to 1023 scans/frame, odd counts only. 2) in non-interlaced mode—scans/frame = 2X + 256. Therefore for 262 scans, program X = 3 (00000011). Range = 256 to 766 scans/frame, even counts only. In either mode, vertical sync width is fixed at three horizontal scans (≡ 3H).
Vertical Data Start	8 bits defining the number of raster scans from the leading edge of vertical sync until the start of display data. At this raster scan the data row counter is set to the data row address at the top of the page.
Data Rows/Frame	6 bits assigned providing up to 64 data rows per frame.
Last Data Row	6 bits to allow up or down scrolling via a preload defining the count of the last displayed data row.
Scans/Data Row	4 bits assigned providing up to 16 scan lines per data row.

Additional Features

Device Initialization:

Under microprocessor control—The device can be reset under system or program control by presenting a 1Ø1Ø address on A3-Ø. The device will remain reset at the top of the even field page until a start command is executed by presenting a 111Ø address on A3-Ø.

Via "Self Loading"—In a non-processor environment, the self loading sequence is effected by presenting and holding the 1111 address on A3-Ø, and is initiated by the receipt of the strobe pulse ($\overline{DS}$). The 1111 address should be maintained long enough to insure that all seven registers have been loaded (in most applications under one millisecond). The timing sequence will begin one line scan after the 1111 address is removed. In processor based systems, self loading is initiated by presenting the Ø111 address to the device. Self loading is terminated by presenting the start command to the device which also initiates the timing chain.

Scrolling—In addition to the Register 6 storage of the last displayed data row a "scroll" command (address 1Ø11) presented to the device will increment the first displayed data row count to facilitate up scrolling in certain applications.

Description of Pin Functions

Pin No.	Symbol	Name	Input/ Output	Function
25-18	DBØ-7	Data Bus	I/O	Data bus. Input bus for control words from microprocessor or PROM. Bidirectional bus for cursor address.
3	CS	Chip Select	I	Signals chip that it is being addressed
39,40,1,2	AØ-3	Register Address	I	Register address bits for selecting one of seven control registers or either of the cursor address registers
9	$\overline{DS}$	Data Strobe	I	Strobes DBØ-7 into the appropriate register or outputs the cursor character address or cursor line address onto the data bus
12	DCC	DOT Counter Carry	I	Carry from off chip dot counter establishing basic character clock rate. Character clock.
38-32	HØ-6	Character Counter Outputs	O	Character counter outputs.
7, 5, 4	R1-3	Scan Counter Outputs	O	Three most significant bits of the Scan Counter; row select inputs to character generator.
31	H7/DR5	H7/DR5	O	Pin definition is user programmable. Output is MSB of Character Counter if horizontal line count (REG.Ø) is ≥128; otherwise output is MSB of Data Row Counter.
8	RØ	Scan Counter LSB (Odd/Even Field)	O	Least significant bit of the scan counter. In interlaced mode this bit defines the odd or even field. In this way, odd scan lines of the character font are selected during the odd field and even scans during the even field.
26-30	DRØ-4	Data Row Counter Outputs	O	Data Row counter outputs.
17	BL	Blank	O	Defines non active portion of horizontal and vertical scans.
15	HSYN	Horizontal Sync	O	Initiates horizontal retrace.
11	VSYN	Vertical Sync	O	Initiates vertical retrace.
10	CSYN	Composite Sync	O	Active in non-interlaced mode only. Provides a true RS-170 composite sync waveform.
16	CRV	Cursor Video	O	Defines cursor location in data field.
14	V_{CC}	Power Supply	PS	+5 volt Power Supply
13	V_{DD}	Power Supply	PS	+ 12 volt Power Supply

Timing Diagrams

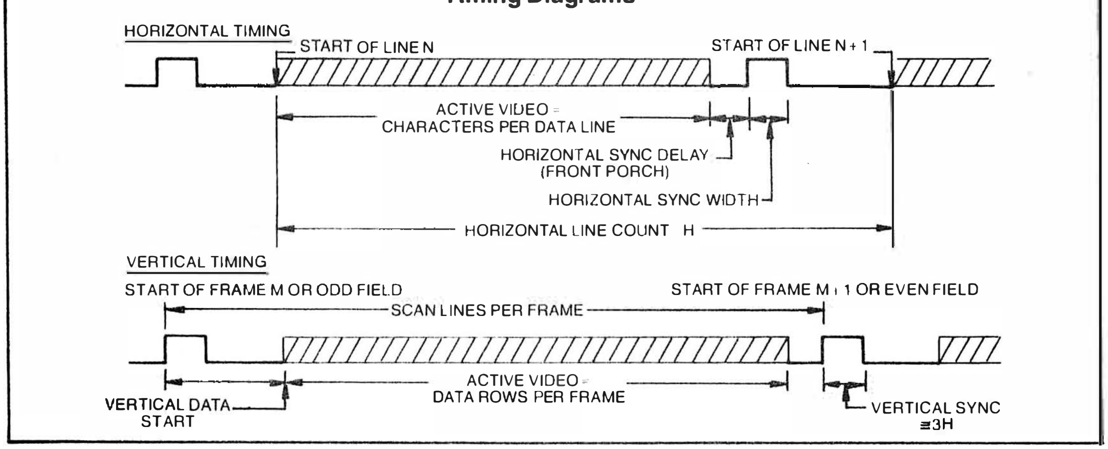
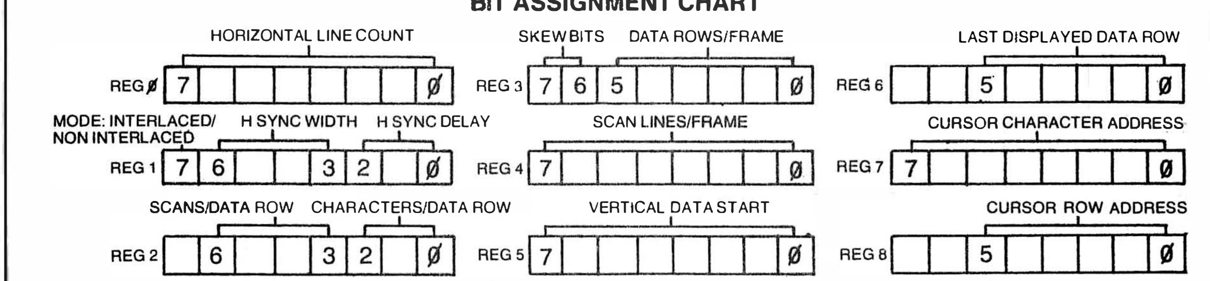


TABLE 1

BIT ASSIGNMENT CHART

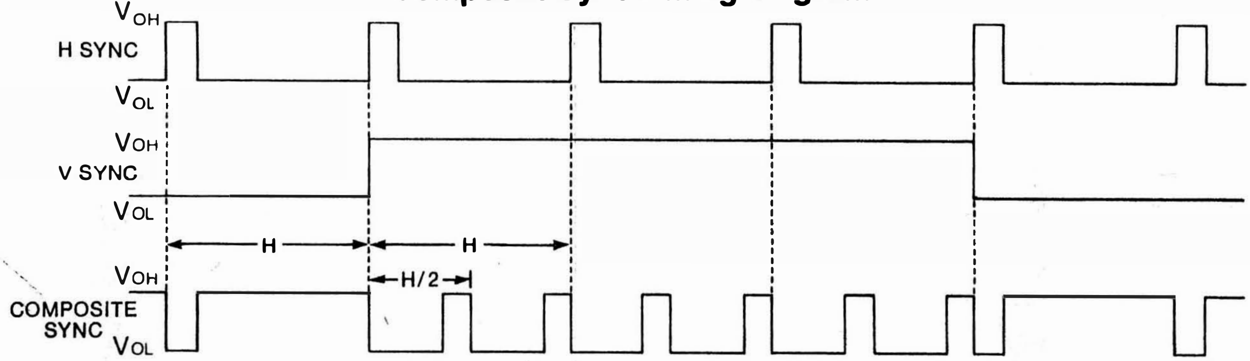


Register Selects/Command Codes

A3	A2	A1	AØ	Select/Command	Description
0	0	0	0	Load Control Register Ø	See Table 1
0	0	0	1	Load Control Register 1	
0	0	1	0	Load Control Register 2	
0	0	1	1	Load Control Register 3	
0	1	0	0	Load Control Register 4	
0	1	0	1	Load Control Register 5	
0	1	1	0	Load Control Register 6	
0	1	1	1	Processor Self Load	Command from processor instructing CRT 5027 to enter Self Load Mode
1	0	0	0	Read Cursor Line Address	
1	0	0	1	Read Cursor Character Address	
1	0	1	0	Reset	Resets timing chain to top left of page. Reset is latched on chip by $\overline{DS}$ and counters are held until released by start command.
1	0	1	1	Up Scroll	Increments address of first displayed data row on page. ie; prior to receipt of scroll command—top line = 0, bottom line = 23. After receipt of Scroll Command—top line = 1, bottom line = 0.
1	1	0	0	Load Cursor Character Address*	
1	1	0	1	Load Cursor Line Address*	
1	1	1	0	Start Timing Chain	Receipt of this command after a Reset or Processor Self Load command will release the timing chain approximately one scan line later. In applications requiring synchronous operation of more than one CRT 5027 the dot counter carry should be held low during the $\overline{DS}$ for this command.
1	1	1	1	Non-Processor Self Load	Device will begin self load via PROM when $\overline{DS}$ goes low. The 1111 command should be maintained on A3-Ø long enough to guarantee self load. (Scan counter should cycle through at least once). Self load is automatically terminated and timing chain initiated when the all "1's" condition is removed, independent of $\overline{DS}$. For synchronous operation of more than one CRT 5027, the Dot Counter Carry should be held low when this command is removed.

*NOTE: During Self-Load, the Cursor Character Address Register (REG 7) and the Cursor Row Address Register (REG 8) are enabled during states Ø111 and 1ØØØ of the R3-RØ Scan Counter outputs respectively. Therefore, Cursor data in the PROM should be stored at these addresses.

Composite Sync Timing Diagram



AC TIMING DIAGRAMS

FIGURE 1 VIDEO TIMING

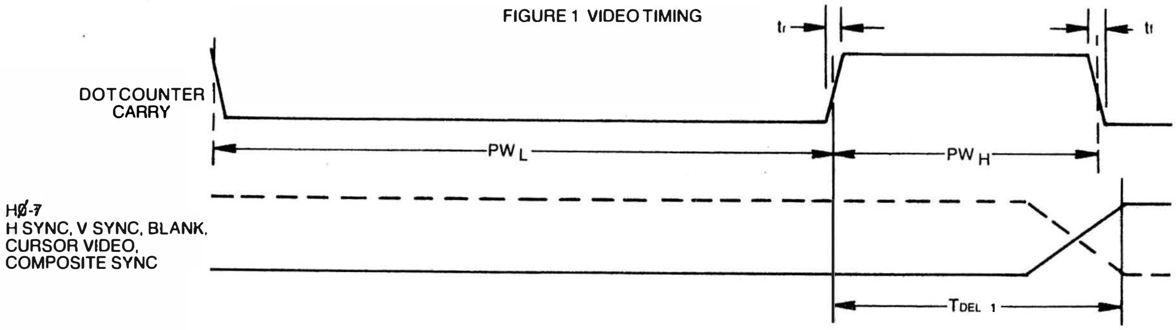


FIGURE 2 LOAD/READ TIMING

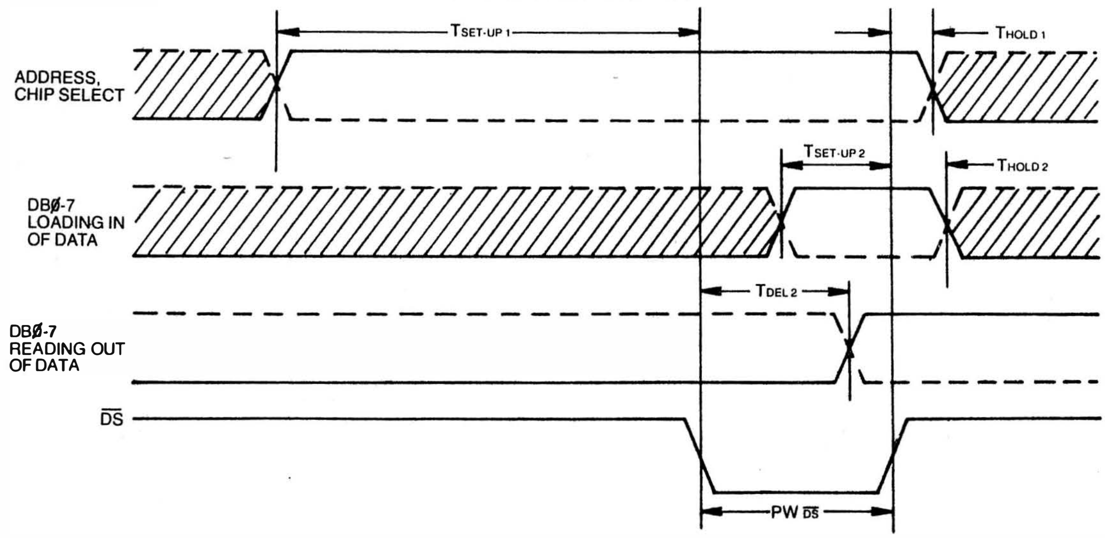


FIGURE 3 SCAN AND DATA ROW COUNTER TIMING

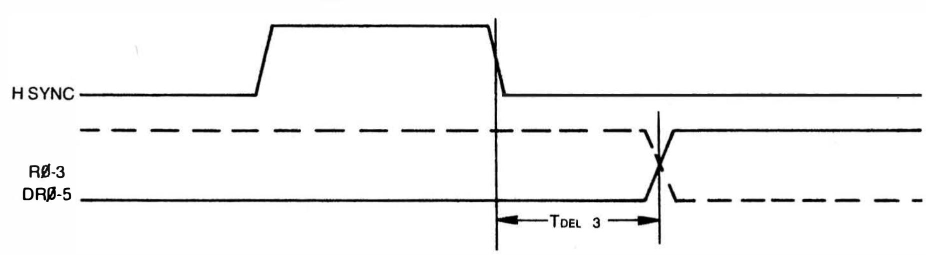
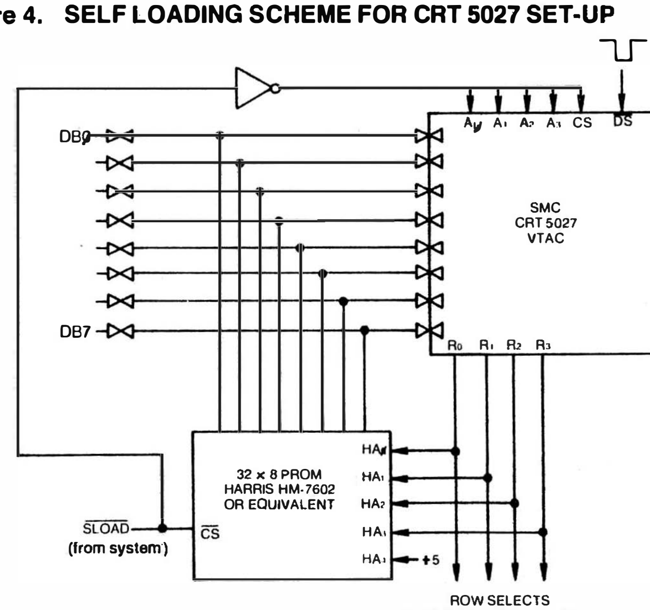
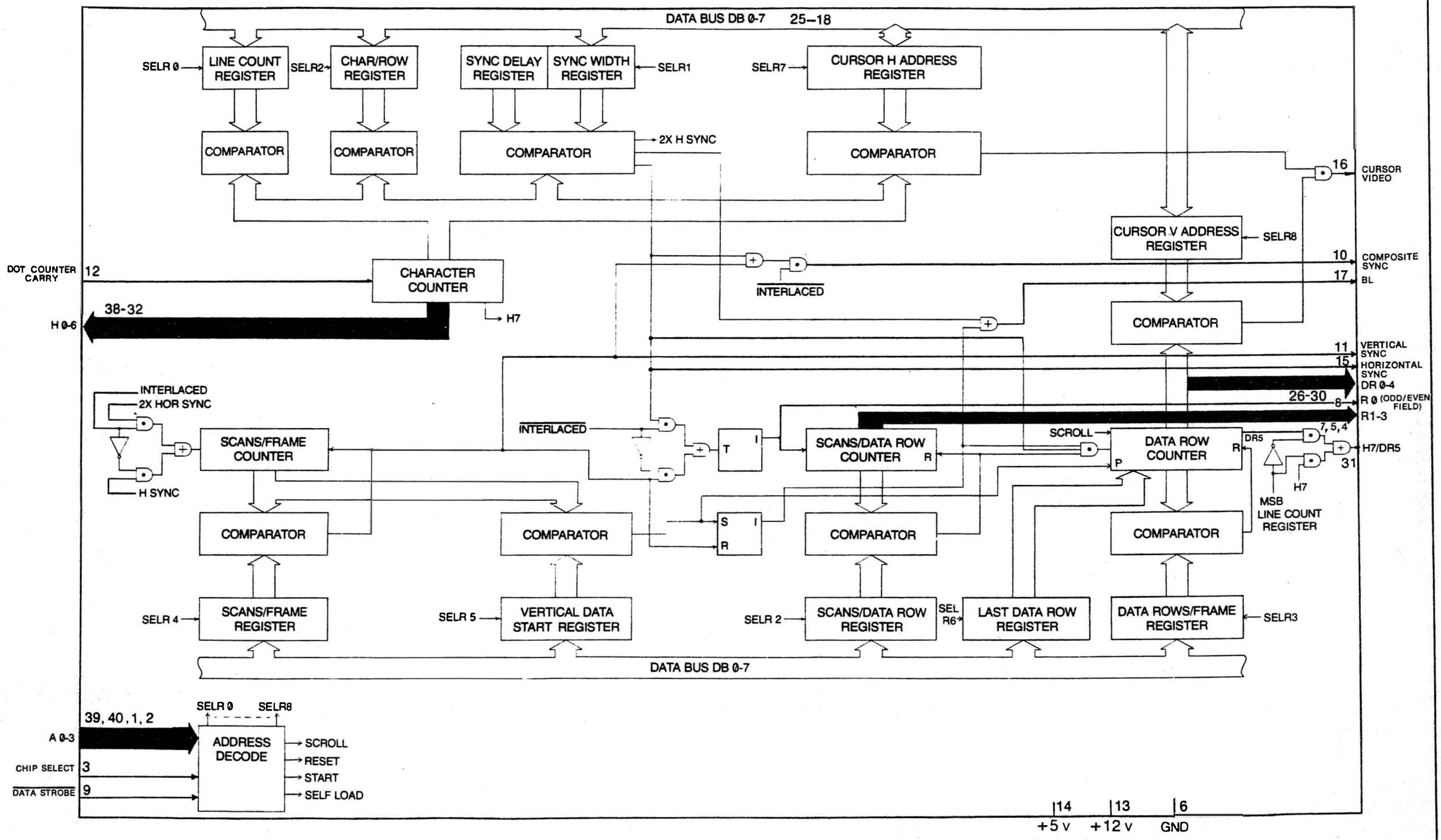


Figure 4. SELF LOADING SCHEME FOR CRT 5027 SET-UP





BLOCK DIAGRAM

CRT 5027 Control Registers Programming Chart

Horizontal Line Count: Total Characters/Line = N+1, N=0 to 255 (DBØ=LSB)

Characters/Data Row:

DB2	DB1	DBØ		
0	0	0	= 20	Active Characters/Data Row
0	0	1	= 32	
0	1	0	= 40	
0	1	1	= 64	
1	0	0	= 72	
1	0	1	= 80	
1	1	0	= 96	
1	1	1	= 132	

Horizontal Sync Delay: =N, from 1 to 7 character times (DBØ=LSB) (N=0 Disallowed)

Horizontal Sync Width: =N, from 1 to 15 character times (DB3=LSB) (N=0 Disallowed)

Skew Bits

DB6	DB7	Sync/Blank Delay (Character Times)	Cursor Delay (Character Times)
0	0	0	0
0	1	1	0
1	0	2	1
1	1	2	2

Scans/Frame

8 bits assigned, defined according to the following equations: Let X = value of 8 assigned bits. (DBØ=LSB)

1) in interlaced mode—scans/frame = 2X + 513. Therefore for 525 scans, program X = 6 (00000110). Vertical sync will occur precisely every 262.5 scans, thereby producing two interlaced fields.
Range = 513 to 1023 scans/frame, odd counts only.

2) in non-interlaced mode—scans/frame = 2X + 256. Therefore for 262 scans, program X = 3 (00000011).
Range = 256 to 766 scans/frame, even counts only.

In either mode, vertical sync width is fixed at three horizontal scans (≡3H).

Vertical Data Start: N= number of raster lines delay after leading edge of vertical sync of vertical start position. (DBØ=LSB)

Data Rows/Frame: Number of data rows = N+1, N=0 to 63 (DBØ=LSB)

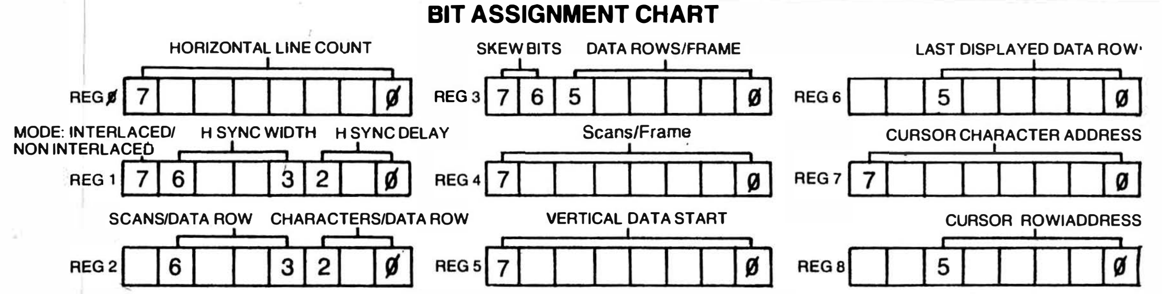
Last Data Row: N= Address of last displayed data row, N=0 to 63, ie; for 24 data rows, program N=23. (DBØ=LSB)

Scans/Data Row: =N+1, N=0 to 15 (DB3=LSB)

Mode: DB7=1 establishes interlace

TABLE 1

BIT ASSIGNMENT CHART



SMC Microsystems Corporation 35 Marcus Boulevard Hauppauge, New York 11787 (516) 273-3100 TWX: 510-227-8898 1177 10K