POWERTRAN CORTEX

CORTEX POWER BASIC AND ASSEMBLY LANGUAGE MONITOR USER'S GUIDE

CONTENTS

This page is intentionally blank

1. INTRODUCTION

This reference guide contains Information about the software supplied with the Cortex microcomputer. This consists of Power Basic and an assembly language monitor.

Although this user's guide is not intended as a beginner's tutorial, sections 2 and 3 do give a summary of programming in Basic.

Section 4 explains how each Basic keyword is used and examples are given where appropriate. Detailed information is given for some statements, such as how variables are stored in memory. This is given for those who require a deeper understanding of the system. The user can, however, write programs effectively without delving into the inner workings of Basic.

Section 5 explains how to use the color graphics statements and section 6 deals with character strings. These are used a little differently in CORTEX BASIC from other computers.

The remainder of the book, sections 7 and 8, is intended for the advanced user who wishes to program in assembly language and do low level I/O.

A reference section is included at the back of the book which brings together a lot of information in compressed form. This section will be useful when you are familiar with using your CORTEX computer.

This user's guide refers to memory in terms of BYTES and WORDS. The CORTEX is a 16-bit microcomputer. This means that information is handled in blocks of 16 binary digits. This block is known as a WORD. Users of 8 bit computers may be more familiar with the BYTE, which is 8 bits long. On the CORTEX, one WORD equals two BYTES.

The expression "Kbyte" is short for Kilobyte and means 1024 bytes. This is often abbreviated to just "K". The CORTEX is supplied with 64K of RAM, which works at 65536 bytes. Some of this is used by BASIC; the rest is free for your programs.

This page is intentionally blank

2. THE BASIC LANGUAGE

2.1 Introduction to the BASIC Language

This section gives a brief introduction to the BASIC language and how to use it. If you are already familiar with BASIC, you can skip to section 2.2.

When first turned on, the CORTEX puts the following message on the television screen.

```
CORTEX BASIC Rev. x.x (C) 1982
*Ready
```

"x.x" gives the version of BASIC that you have.

Check that the CAPS LOCK key is pressed down. Type in SIZE, followed by the "RETURN" key. BASIC will respond with details of the memory space allocated to program end variable storage and the amount of memory left:

```
SIZE
PRGM:10 Bytes
VARS:0 Bytes
FREE:34174 Bytes
```

The actual numbers will depend on how much memory there is in the system and on the version of BASIC.

SIZE is an example of a BASIC command. It is carried out immediately when "RETURN" is pressed. Commands include things like listing, storing, loading and executing BASIC programs. A full list of commands is given in appendix A of the reference section.

Apart from commands, the other type of input that can be entered from the keyboard is a statement. A statement is an instruction to do something, but it is not carried out immediately. The following statement is an instruction to add two numbers together and to print the result:

```
30 PRINT A+B
```

Type it on your CORTEX exactly as shown end press "RETURN".

> If you make a mistake typing, there are two things you can do: (1) press the "RUBOUT" key to backspace, one by one, over the characters that are wrong, and then type them again; or (2) press the escape key (marked "ESCAPE"), and then type the whole line again (Section 3 gives fuller details of editing lines).

BASIC will perform some checks on the line after you have pressed "RETURN". If it discovers an error it will print an appropriate error message, print the line on the screen again and position the cursor where it thinks the error is, so that you can type over it. Here is an example:

```
30 PRINT A.8
** Illegal character **

30 PRINT A.B
          ^
          |
Cursor positioned here
```

Overtype the mistake with the correct character, and then press "RETURN". Remember, you can use the "RUBOUT" key to backspace over the line, or press "ESCAPE" to type the whole line again.

A statement is normally preceded by a line number (30 in the example above) to distinguish it from a command. Statements with a line number are not carried out immediately; they are stored in memory for future use. Several statements can be entered and stored, and then executed together as a program. The line numbers determine the order in which the statements will be carried out. It does not matter in which order the statements were actually entered: BASIC arranges them into the correct order.

In the statement just entered, "A" and "B" are variables: they stand for values which can be set up, referred to, and changed by program statements. A and B represent storage locations in the computer memory.

The types of values that can be used in BASIC programs are

- Integers (whole numbers 1, 2, 3, etc) up to 32,767 and down to -32,768.
- Floating point numbers (i.e. decimals such as 12.34 and 0.001234)
- Strings (i.e. sequences of characters enclosed in quotes - "THIS IS A STRING")

Variables are described more fully in section 2.7

As it happens, BASIC cannot execute statement 30 as it is, because A and B do not yet have any values to add together or print. Variables must be given a value before they can be used. If you tried to execute statement 30 as it is, BASIC would print an error message:

```
** Undefined variable ** at 30
```

Until a variable has been assigned a value, BASIC does not even know of the variable's existence and is unable to do any calculations with it.

To resolve this difficulty, two more statements can be added to the program with appropriate line numbers to ensure that they are executed before statement 30:

```
10 A=5
20 INPUT B
```

Statement 10 is an assignment statement that introduces the variable A to BASIC and assigns the value 5 to it. Once this statement has been executed A can be referred in any later statements and will have the value 5 (unless it is changed in the meantime). Further assignment statements can be used to change the value of A.

Statement 20, when executed, introduces the variable B and causes BASIC to ask for a value for it. Having asked, it waits until a number is entered from the keyboard.

Note that neither of these things has happened yet. What you have done is to type in some statements which make a program that will be executed when we give BASIC the command to do so. Despite having entered the new statements, A and B are still undefined - and will continue to be undefined until we RUN the program.

To look at the program you have just entered type

```
LIST
```

followed by "RETURN". LIST is another command like SIZE. BASIC will list the program stored in its memory in the order in which it w111 be executed:

```
10 A=5
20 INPUT B
30 PRINT A+B
```

If your listing does not look like this, re-type the line that is wrong. If you have typed a wrong line number, remove it from the program by typing the line number followed by "RETURN". Then you can enter the correct line. Again use LIST to check that the program is correct.

So far, we have simply constructed a program - a list of statements - in BASIC's memory. To carry out (execute) this program, type the command

```
RUN
```

(followed by "RETURN").

BASIC will perform statement 10, and give a value of 5 to the new variable A. You will not observe anything while this is happening.

Halfway through executing statement 20, BASIC will output a prompt (?) to the screen and will wait for you to enter a value for the new variable B. Type a number followed by "RETURN". BASIC will add the values of A and B together, print the result, and return to wait for the next command. The whole sequence will look something like this:

```
RUN
? 256
261

Stop at 30
```

"256" is the value entered for "B"; BASIC has printed "261" which is 256 plus the value assigned to A. Try RUN again, this time entering a different number, perhaps one with a decimal point.

You can also change the program and re-run it. You could set A to a different value, or replace "A+B" with a different mathematical formula. As with most computer languages, "*" is used for multiply and "/" for divide; "+" and "-" have the usual meaning. See section 4 for a full description.

> You can change, or edit, the program simply by retyping the appropriate line, or adding a new one with a suitable line number. BASIC also provides a simple line editor that allows single characters to be changed, inserted or deleted without retyping the whole line. Section 3 gives the commands available for editing single lines.

Note that the assignment statement "A=5" is not simply a statement of fact as it would be in mathematics ("A equals 5"). In BASIC it stands for the action: "make A equal to 5". Thus a statement such as "A=A+5" is allowed in BASIC (provided A has already been assigned a value); try it, and see what it does.

The assignment statement is sometimes also called the LET statement in BASIC, because it can also be written:

```
15 LET A=A+5
```

The LET part is optional and is usually omitted.

One useful thing to note is that a statement entered without a line number is not stored, but is carried out immediately, like a command. If you want to know the internally stored value of a variable at any time, simply type

```
PRINT A (or any other variable that you want to see)
```

PRINT is used so frequently that it can be abbreviated to one character, a semicolon or question mark. The above statement can also be written

```
;A or ?A
```

The following is a more complex program which can be used to explore some of the additional features of BASIC. To prepare for entering this program, first type the command

```
NEW
```

which will clear all statements and variables from BASIC's memory ready for a new program. Now type the following:

```
10 DIM $A(4)
20 $A(0)="THE NUMBER IS"
30 INPUT "INPUT NUMBER", N
40 IF N-INT(N)<>0 THEN PRINT $A(0);N;: GOTO 60
50 GOSUB 100 ! EVEN OR INTEGER
60 PRINT ", ITS SQUARE IS";N*N;", AND ITS SQUARE ROOT IS";
70 IF N<0 THEN PRINT " UNDEFINED": GOTO 30
80 PRINT SQR(N)
90 GOTO 30
100 IF INT(N/2)*2=N THEN PRINT $A(0);" EVEN"; :RETURN
110 PRINT $A(0);" ODD";
120 RETURN
```

Correct any typing errors as described above. You must get all the punctuation exactly right. BASIC will find some errors. If you leave out one quotation mark (") it will be detected, because quotation marks always come in pairs surrounding a length of text - "THE NUMBER IS". Other errors may not be detected until you RUN the program.

LIST the program as a final check. You may notice that line 10 is listed as

```
      10 DIM $A[4]
```

This is perfectly correct; BASIC makes no distinction between parentheses () and square brackets []. Your program is stored by BASIC in a condensed and encoded form to take up as little space as possible. (For this reason, BASIC programs are very compact). When you enter a statement, a part of BASIC called the editor takes your input and converts it into this coded form. The editor is also responsible for translating the coded program back into an understandable form and printing it when you type LIST. While you are entering, changing or listing program statements you are using the editor, though you will normally be unaware of this.

When you type RUN, another part of BASIC called the interpreter takes over from the editor. The interpreter fetches the stored program statements one by one and carries out the coded instructions.

RUN the new program and enter a number (followed by "RETURN") in response to the question mark prompt. Press the ESCAPE key to get out of the program. (This will work at any time, even if BASIC is waiting for an input. It is the normal way of stopping a program that seems unwilling to stop by itself). The result should be something like this:

```
RUN
  INPUT NUMBER? 17
  THE NUMBER IS ODD, ITS SQUARE IS 289, AND ITS SQUARE ROOT IS 4.1231056256
  INPUT NUMBER? -6
  THE NUMBER IS EVEN, ITS SQUARE IS 36, AND ITS SQUARE ROOT IS UNDEFINED
  INPUT NUMBER, 2.35
  THE NUMBER IS 2.35, ITS SQUARE Is 5.5225, AND ITS SQUARE ROOT IS 1.5329709716
  INPUT NUMBER? (escape key)
STOP AT 30
```

There are several things to note about this program:

- "$A" is a string variable which holds text rather than numeric values. Where a variable is used to store strings, its name is preceded by a "$" (lines 20, 40, 100, 110). String variables are described in section 2.5.5.

- $A is also an array (sometimes known as a dimensioned variable), which means that it stands not for a single storage location but for several storage locations grouped together in memory. Arrays are often used to store text strings which usually require more space than is available in a single storage location. They can also be used to store groups of numeric values. Statement 10 (the DIMension statement) tells BASIC that $A is going to be an array rather than an ordinary variable and also declares how big it is. Arrays are described in section 2.5.6, and some additional information on string arrays is given in section 6.

- The INPUT statement can specify a text prompt to be printed before the question mark (line 30). There are many more options for the INPUT statement, see section 4.

- The IF .. THEN statement (line 40) allows conditional execution of an action. This statement means "IF the number just entered is not a whole number THEN print the string stored in the array $A, followed by the number N, and go to line 60 for the next statement" (otherwise do nothing, and continue with line 50). "<>" means "is not equal to"; INT is described below.

- It is possible to place more than one BASIC statement on a single line, separated by ":" (lines 40, 70, 100). Each Statement is executed in order. This feature is particularly useful with IF .. THEN statements: all statements on the same line after the THEN will be executed if the condition is true and not executed if it is false.

- Comments can be placed at the end of a statement line. They are preceded by "!". Everything after the exclamation mark is ignored by BASIC when it executes the statement (line 50). However comments do take up storage apace in program memory.

- Statements such as GOTO and GOSUB can be used to alter the normal flow of program execution which is in order of increasing line number. The GOTO statement in line 40 w111 cause the BASIC interpreter to go directly to line 60 to find the next statement. The GOSUB statement in line 50 causes similar transfer to line 100. In this case BASIC "remembers" where it was; when the interpreter finds the RETURN statement in line 120, it goes back to the end of the GOSUB statement (line 50), and executes line 60 next. The GOTO is a permanent diversion of program flow; the GOSUB is a temporary diversion which will return to the main program. Statements 100 to 120, which are executed out of the main line of the program under control of the GOSUB statement, are called a subroutine.

- The PRINT statement (lines 40, 60, 70, 100, 110) has a number of options to allow output of text and layout of printed numbers on the page. (Try substituting "," for ";" in one of the print statements, and see what happens).

- SQR (square root) and INT (integer part) are two examples of the standard functions available in BASIC. It is also possible to define new functions in a program (section 4).

- This program, as it is written, is an endless loop: there is no way to exit from the program except by pressing ESCAPE.

By understanding and experimenting with this program, you will learn some of the more important features of BASIC. Section 2.2 gives further information about BASİC in a more formal and systematic way than is presented above. In section 4 there are descriptions of the individual commands, statements and operations that make up the BASIC language. Appendices A and B give a complete Quick Reference Guide to CORTEX BASIC.

Having written a BASIC program, you may wish to save it on cassette tape for more permanent storage. Section 3 describes how to do this.

For special purposes, subroutines can be written in 9995 assembly language and called up from a BASIC program.

2.2 CORTEX BASIC Program

A CORTEX BASIC program consists of a number of statement lines. Each line is preceded by a line number which must be an integer in the range 1 to 32767. This line number indicates the order in which the statement lines w111 be arranged before the program is executed. The lowest line number will be the first line and the highest line number will be the last line. These statements may either perform some action such as adding two variables together and assigning the sum to a third variable (e.g. 'A=B+C'), or may be control statements such as GOSUB 3000 that change the order of execution of the program.

To save memory a number of statements can be written on one line using the statement concatenation operator (:). Each statement will be executed in turn. The general syntax for a line is:

```
{ line number } statement [ : statement ]  { ! comment }
```

where `{ }` indicates optional items
`[ ]` indicates item is repeated as many times as required.

Exceptions:

- DATA must be the only statement on a line.
- DEF must be the only statement on a line.
- ON must be the only statement on a line.
- NEXT should be the first statement on a line, otherwise it may not be found to terminate its corresponding FOR loop.
- REM makes the remainder of the line a comment.
- STOP must be the last statement on a line.

A full list of the CORTEX BASIC commands/statements is provided in section 4.

2.3 Character Set

1) Upper and lower case alphabet.
2) Digits 0 to 9.
3) Special characters
   ! " # $ % ^ ' ([ ]) * : = - + ; , . ? / < >

Note: The THEN keyword can be abbreviated to ':' and PRINT can be abbreviated to ';' or '?'.

2.4 Constants

Constants are entered as decimal or hexadecimal numbers. A decimal constant uses the digits 0 to 9 and "." for the decimal point. E is used for 'times ten to the power of', e.g. 4000 could be written as 4E3.

A hexadecimal constant is one to four hex digits followed by the letter H. A hex constant beginning with one of the letters A - F must be preceded by a zero.

2.5 Variables

A CORTEX BASIC variable can be used to store either an integer number, a real (floating point) number, or a character string depending on the context in which the variable is used. Although a variable may contain a number (integer or real) it can be used as though it contained a character string, and vice versa.

Variable storage starts in high memory and builds down towards low memory as new variables are declared, with each variable being allocated six consecutive bytes of memory. A variable's address is that of the word lowest in memory, i.e. the word nearest to address zero. In the diagrams this is referred to as the 'first word'.

2.5.1 Variable Names

A variable name is either an alphabetic character followed by a number in the range 0 to 127 (e.g. Z100) or an alphabetic string up to three characters long (e.g. A, ST, and LST). The variable name cannot be identical to a CORTEX BASIC keyword, nor can it form the beginning of a keyword. The following variable names are not valid:

LIS	Beginning of LIST (a BASIC statement)
MEM	A BASIC function
TOT	First 2 letters are the BASIC keyword TO
12B	First character is not alphabetic
ABCD	More than 3 characters
I130	Number greater than 127
A.B	'.' not allowed in variable names

Note: There is a maximum of 140 different variable names in any one CORTEX BASIC program.

2.5.2 Variable Declarations

Variables do not have to be declared at the beginning of a program. They are allocated space in memory the first time you assign a value to them. For example, to declare the variable TST and assign to it the value 100 the following statement can be used:

```
TST=100
```

A value can be assigned to a variable by either a READ (read a value from a DATA statement), an INPUT (accept input from the terminal) or a LET statement. The statement 'TST=100' is an implied LET, as are all statements of the form:

```
<variable>=<expression>
```

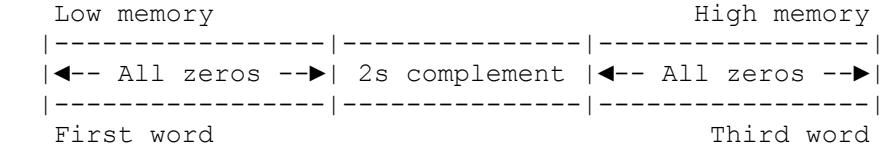
where <expression> may contain function calls:

```
FRD=SIN(PI*NUM)
```

The above statement assumes that the variables PI and NUM have already been given a value.

2.5.3 Integer Variable

If a number can be represented in a 16-bit twos complement form, it is stored in integer format. Integer numbers in the range -32768 to +32767 will be stored like this:

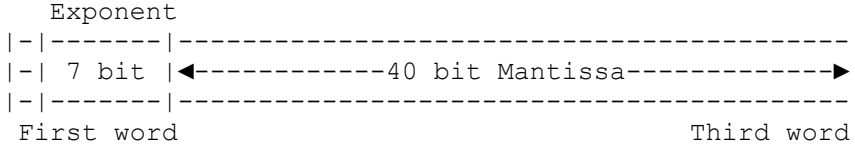


Integer numbers outside this range (up to approximately 11 decimal digits) will be recognized as integers and printed out in integer format, even though they will be stored internally in floating point format (see below). Integer numbers greater than this will be treated as real numbers and will be printed out in floating point format (e.g. 1.234E11).

2.5.4 Floating Point Variables

Floating point format allows a real number in the range 0E-75 to 10E+74 to be stored. 'E' represents the multiplier 10; the integer number following is the power which 10 raised. 2.5E24 means 2.5 times 10 to the power 24. Real constants can be entered in this format if desired. This representation provides approximately 11 decimal digits of accuracy and so a real constant should consist of no more than 11 digits.

A floating point number is represented internally as a fraction multiplied by a power of 16 (this power is known as the exponent), and is stored as:

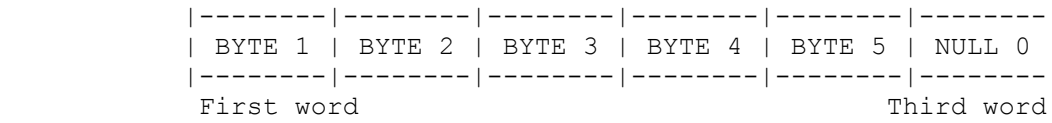


Bit 0 (the most significant bit) is the sign bit and represents the sign of the floating point number: 0 for positive, 1 for negative. Bits 1 to 7 hold the exponent coded in Excess 64 notation (the exponent is incremented by 64; this gives the exponent a range of 0 to 127 representing a true exponent range of -64 to +63). The remaining 40 bits contain the normalized mantissa (the mantissa is normalized if its first hex digit is non-zero). Negative fractions are stored in true form with the sign bit set to one and not in twos complement notation.

2.5.5 Character String Variables

A character string is a sequence of characters enclosed within single or double quotes. If you wish to use single quotes within a string, the string must be enclosed by double quotes and vice versa.

When preceded by a dollar sign ($), the variable is recognized as containing a string. In this form, a variable can be used to store up to 5 characters. The last of the 6 bytes is used to terminate the string and contains the null character (zero). This is necessary to ensure that the variable defined immediately before the string variable does not get overwritten and corrupted.



Non-printable characters may be included in a character string by writing their hexadecimal ASCII representation enclosed in angle brackets (<>). The angle brackets are stored along with the character string and are only interpreted when the string is being input from a terminal, read from a DATA statement, or when the string is being printed. Note: Attempting to use the character sequence '<>' in a string via an INPUT, READ or PRINT statement will cause problems. If these characters are required then the sequence '<3C><3E>' should be used.

2.5.6 Array Variables

An array is a number of variables (stored consecutively in memory) that is referenced by a single variable name. Individual variables (or array elements) are accessed by following the variable name with a number that identifies the position of the variable within the array. This number - the array subscript - is enclosed in parentheses or square brackets. Parentheses will be converted into square brackets by BASIC.

To allocate 10 elements to the array ARR the following statement is required:

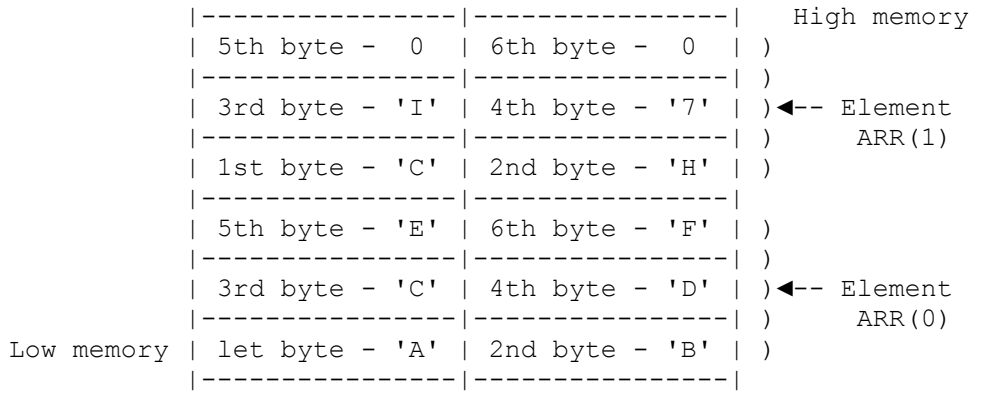
```
DIM ARR(9)
```

The elements are referenced by

```
ARR(0), ARR(1), ...., ARR(9).
```

The size parameter supplied to the DIMension statement is one less than might be expected, as CORTEX BASIC automatically allocates space starting from element zero.

An array may also be used to hold character strings. Individual bytes of a character string array can be accessed by following the array subscript with a semicolon (;) and the number of the required byte starting from 1. For example, $ARR(1;3) references the third byte of array element ARR(1); this corresponds to the letter 'I' in the diagram below, which shows storage of the string "ABCDEFGHIJ".



CORTEX BASIC allows an array to be declared with any number of dimensions. For most practical applications a two dimensional array is usually sufficient.

Note: The variable A and the array variable A(0) refer to two completely different variables.

2.6 Operators and Expressions

An expression is a list of variables and constants, separated by operators. There are three types of POWER BASIC operators and expressions: arithmetic, logical, and relational.

2.6.1 Arithmetic Operators

The following is a list of the valid arithmetic operators:

```
+ addition
- subtraction
* multiplication
/ division
^ exponentiation
+ unary plus
- unary minus
```

2.6.2 Arithmetic Expressions

An arithmetic expression is any valid sequence of numbers, variables, and binary operators (properly balanced, no two numbers or variables can be adjacent, and no two binary operators can be adjacent).

For example:

An expression may consist of a single operand:

```
8
SIN(A)
```

A sequence of operands may be combined by arithmetic operators:

```
X*Y
A*B-W/Z
```

Any expression may be enclosed in parentheses and considered to be a basic operand:

```
(X+Y)/Z
(A+B)*(C-D)
```

Any expression may be preceded by a plus or minus sign:

```
+X
-(A+B)
-A+((TAN(-A))*2)
```

2.6.3 Logical Operators

The logical operators do "bit-wise" operations on integers. They consist of the following:

```
LNOT (unary) 1's complement of integer
LAND (binary) Bit-wise AND of two integers
LOR  (binary) Bit-wise OR of two integers
LXOR (binary) Bit-wise exclusive OR
```

2.6.4 Logical Expressions

Logical expressions are similar to arithmetic expressions. They both consist of variables, constants, parenthesis, and operators. The primary difference is that the operators are different for logical expressions. The logical operators perform a bit-wise logical operation on the operand(s). For example, if A = 0AAAAH (hex "AAAA"), and B = 05555H (hex "5555") and C = 0BBBBH, (hex "BBBB"), then

```
LNOT (A)    would equal  "5555"
A LAND B    would equal  0
A LOR B     would equal  'FFFF'
A LXOR C    would equal  '1111'
```

2.6.5 Relational Operators

The relational operators are all binary operators that operate on two arithmetic expressions. They return values of 1 (TRUE) or 0 (FALSE). Relational operators consist of the following:

```
=     exactly equal
==    approximately equal (plus or minus 9.5 E-07)
<     less than
<=    less than or equal to
>     greater than
>=    greater than or equal to
<>    not equal
```

The approximately equal (==) relational operator returns a TRUE value when the absolute value of the difference between the two values is less than or equal to 9.5 E-07.

2.6.6 Boolean Operators

The Boolean operators are designed to work on the resultant TRUE or FALSE conditions set by the relational operators. However, they may also operate on variables within the program, in which case a zero value is considered False and a non-zero value variable is considered to be True. The Boolean operators return values of 1 (True) or 0 (False).

Boolean operators consist of the following:

NOT	(UNARY)	Returns a TRUE value if expression evaluates to FALSE (non-zero); otherwise, returns a FALSE value.
AND	(BINARY)	Returns a TRUE value if both expressions evaluate to TRUE (non-zero); otherwise, returns a FALSE value.
OR	(BINARY)	Returns a TRUE value if either expression evaluates to TRUE (non-zero); otherwise, returns a FALSE value.

2.6.7 Boolean and Relational Expressions

Boolean and relational expressions are formed according to the following rules:

A Boolean or relational expression may consist of a single element:

```
NOT(A)
X<>3.10159
```

Single elements may be combined through the use of the Boolean operators AND and OR to form compound expressions such as:

```
A AND B
X OR Y
```

Any expression may be enclosed in parentheses and regarded as an element:

```
(T OR S) AND (R OR Q)
```

2.6.8 Expression Evaluation

Expressions are evaluated left to right if the operators are of equal precedence, and there are no parentheses. It there are parentheses in the expression, the sub-expression within the innermost parentheses is evaluated first. Not all operators have equal precedence – operands which are operated on by an operator of high precedence are evaluated before operations of low precedence.

The precedence of operators is:

1. Expressions in parentheses
2. Exponentiation and negation
3. *,/
5. +,-
6. <=,<>
7. >=,<
8. ==,LXOR
9. NOT,LNOT
10. AND,LAND
11. OR,LOR
12. (=) ASSIGNMENT

3. ENTERING, EDITING AND SAVING PROGRAMS

3.1 Entering Programs

Commands and programs may be typed in when the computer says:

```
*Ready
```

To enter program lines, simply type the line number followed by the line itself. Finish by pressing the RETURN key.

To avoid typing the line number for every line you can use the NUMBER command. After you press RETURN to enter a line, Cortex will automatically give you the next line number. It will start from line 100 and work in steps of 10.

```
NUMBER
```

and the computer responds:

```
100 (type your line here)
```

If you want to start from a different line number, say 10, then you would type:

```
NUMBER 10
```

If you wanted to have steps of 20 between lines, then type:

```
NUMBER 10,20
```

The first parameter gives the starting line number and the second the step between lines.

If you make a mistake the computer may spot it for you. It will output an error message and print the line again ready for editing. How to do this is explained in the next section.

3.2 Editing Source Lines

One method of modifying (or editing) a line is simply to retype the line. However, CORTEX BASIC also supports a line editor that allows the user to change previously entered source statements. The available edit commands are:

	key to press	effect
	ESC	Cancel input line
	RUBOUT	Backspace and remove character
	RETURN	Enter the edited line
	Left arrow	Backspace the cursor one character
	Right arrow	Forward space the cursor one character
	INS	Insertion mode
	DEL	Delete one character
LN	EDIT	Display the line LN for editing

To start editing, type the line number and press the EDIT key. This will display the line with the cursor at the end. If a mistake has been made while entering a line it will be displayed with the cursor at the point where the mistake was discovered.

The cursor can be moved along the line by pressing the right and left arrow keys. Type over the top of displayed characters to change things. To delete a character press the delete key.

An attempt to forward space past the last character entered, or to backspace beyond the first character in the line will only make the Cortex bleep.

To insert characters, press the insert key. As you type, the rest of the line will move along, creating space for the characters you are typing. This effect is cancelled as soon as any other editing key (arrow, delete or return) is pressed.

When the RETURN key is pressed, all characters displayed are entered, regardless of the position of the cursor.

Entering just a line number followed by a RETURN causes the specified line to be deleted from the stored program. If the specified line does not exist then an error message is output.

Entering a statement with a line number that already exists causes the original statement to be replaced by the new one.

Changing the line number causes a copy of the original to be included in the program with the new line number (the original statement line remains unchanged).

The editor is automatically invoked when an error is found in a line after you type RETURN. This may be disabled using the ERROR command, see section 3.4.

3.3 Saving and Loading Basic Programs

Having entered your program as explained in the previous sections, you may want to save it on an audio cassette. To do this, a cassette recorder should be connected to the DIN socket on the back of the computer. This should go to input and output connections on the recorder which are usually labeled MIC and EAR respectively.

The remote start switch of the recorder, if it has one, may also be connected. The computer will then turn the recorder off and on as required. If you do not have this connected, you will have to do this yourself.

To save the current program, type:

```
SAVE "name"
```

The name is whatever you wish to call the program. It must not be more than 8 characters long, and must be in quotes (single or double). The computer will respond:

```
Auto-run ? (Y/N)
```

If you answer "Y", then the program will run as soon as it is reloaded. If you do not require this, type "N". This facility is normally used for finished programs. If the program is still under development, auto run is not normally specified.

The next prompt is:

```
Cassette ready? (Y/N)
```

Check that the recorder is set to record and ready to go. Answer "Y". If you have the remote start switch connected, the computer will turn on the recorder.

The program will now be saved.

When the operation is complete, the computer will print:

```
*Ready
```

Note that your program in memory is not affected by this operation; a copy is sent to the tape.

To load a program previously saved on an audio cassette, the command is:

```
LOAD "name"
```

For name, fill in the name you specified when SAVEing. Don't forget the quotes. The computer will respond:

```
Cassette ready? (Y/N)
```

Check that the recorder is set to play and ready to go. If you have the remote start switch connected, the computer will turn on the recorder. CORTEX will search the tape for a program of the name you have asked for. It will print the names of any others it finds on the way.

When the load is complete, the message:

```
*Ready
```

will appear. If your program is not found, press ESCAPE and check the tape was in the right place and the name was spelt correctly. More information on LOAD and SAVE is given in section 4.

3.4 Saving and Loading in Source Format

An alternative method for cassette storage is what is known as Source format. The program is stored on tape exactly as it is written. To use this method type:

```
BAUD 3,300
UNIT 3
```

The first command sets up the data transfer speed for device number 3 (the cassette recorder). The second turns on device 3. This means that everything that appears on the screen is copied to the tape. Now type:

```
LIST
```

Before pressing "RETURN" turn on the tape recorder and wait for 2 or 3 seconds to put a leader on the tape. When you press the return key the listing of your program is copied to the tape. When finished, type:

```
UNIT -3
```

to turn the copying off again.

To load a program in source format, type, as before,

```
BAUD 3,300
```

The system now expects input from the cassette. Turn the recorder to play and the listing will be copied back just as if you were typing from the keyboard (only faster!).

NOTE: You may find it more reliable to type the BAUD command as above but not press the RETURN key until the tape has reached the steady tone of the 2 or 3 second leader. This avoids random noise on the tape producing random characters.

Although source storage is much slower than LOAD and SAVE, it does allow parts of programs to be saved. This is achieved by only listing part of the program.

Since tape input is treated by the Cortex as though you were typing it in, if a tape error occurs Cortex will stop accepting lines and present the incorrect line for you to edit. This is a useful check that all lines have loaded correctly.

However if you have a tape with errors, you can load all correct lines by turning off the error editing. This is done by typing the ERROR command. Any lines with errors will be ignored. Typing ERROR again turns the error checking back on.

Programs may also be merged. Loading in source format is just like typing at the keyboard. If we have two programs, one from line 10 to 100 and the other from 200 to 500, loading both one after the other in source format produces a program stretching from 10 to 500.

This method is also used to transfer listings to and from the RS232C interface (or any other device you have added). Simply change the unit number you use. RS232C is unit 2.

The UNIT command may be used to store data files on tape. See UNIT, PRINT and INPUT in section 4.2.7.

4. BASIC REFERENCE GUIDE

This section gives a full explanation of every BASIC keyword.

Keywords are divided into 3 types:

> Commands may only be entered for immediate action, they may not be used in a program.
>
> E.g.: RUN, LIST
>
> Statements may be used as part of a program or for immediate execution. They normally have some parameters which follow the statement.
>
> E.g.: PLOT 1,2 TO 10,10
>
> Functions produce a single value answer. They are used on the right hand side of an equals sign to assign a value to a variable. They may have arguments (in brackets) following the function.
>
> E.g.: A=SIN(1.2)

The keywords in this section are arranged in order of these categories. Within each category they are grouped according to the type of action they perform, such as mathematical or graphical. Appendix A gives an alphabetical list of the keywords and appendix B a list by function. These may be used as references to this section.

The following conventions are used to show the forms of keywords:

Numeric values for command parameters are decimal unless otherwise specified.

Angle brackets, <> , indicate essential elements supplied by the user:

```
10 LET <variable> = <expression>
```

Braces, {} , indicate a choice between alternatives, one of which must be included:

```
10 ON { variable   } THEN GOSUB <statement number list>
      { expression }
```

Brackets, [] , enclose optional items:

```
10 [ LET ] A=4*ATN(1)
```

Items in capitals are entered exactly as shown.

Items in lower case are supplied by the user.

4.1 BASIC COMMANDS

The commands:

4.1.1 NEW

4.1.2 SIZE

4.1.3 LOAD

4.1.4 SAVE

4.1.5 LIST

4.1.6 RUN

4.1.7 CONT

4.1.8 NUMBER

4.1.9 RENUM

4.1.10 PURGE

4.1.11 MON

4.1.12 BOOT

4.1.13 ERROR

4.1.1 NEW

Forms:

```
NEW
NEW <address>
```

The NEW command without an address deletes the current BASIC program and clears all variable space. BASIC responds with "*Ready" and awaits entry of new BASIC programs. The deleted program may be retrieved later if it has been SAVEd.

The memory freed for the new BASIC program starts 1000 bytes above the system software. This leaves 1000 bytes of RAM which are completely unused by BASIC. This may be used for assembly language routines CALLed from BASIC. These may be written using the MONitor. The start of this area is obtained by using the "SYS(7)" function (see section 4).

The form of the NEW command with an address parameter is used to vary the amount of RAM left unused by BASIC. The address is used as the start point for BASIC programs. Typing:

```
NEW SYS(7)
```

will leave no unused RAM, and increases the amount available to BASIC by 1000 bytes.

```
NEW 7000H
```

means that memory from the address given by "SYS(7)" to 6FFFH is free for assembly language use. The memory available to BASIC is correspondingly reduced. An address of less than "SYS(7)" or greater than 0E800H should not be used.

After using the NEW commend with an address, that address becomes the new default value for NEW without a parameter.

Note: The value of "SYS(7)" is currently around 6100H. Future releases of BASIC may use some of the memory up to 6200H. To ensure compatibility some future releases it is recommended that assembly language programs start from an address above 6200H, or are written position independent and loaded at "SYS(7)".

4.1.2 SIZE

Form:

SIZE

The SIZE command prints out to all currently selected output devices the amount of memory taken by the current BASIC program and its variables. The amount of remaining user memory is also printed.

Note: If no user program is currently in memory there will still be a program size. This is the program overhead and is always present. The overhead size is included in the program size so that if the program is burnt into EPROM the user will be able to tell how many EPROMs are required.

EXAMPLE:

```
SIZE
PRGM: 10 Bytes
VARS: 0 Bytes
FREE: 34432 Bytes
```

4.1.3 LOAD

Forms:

```
LOAD "<filename>"
LOAD
```

The LOAD command is used to load BASIC or machine code programs from audio cassette. The filename is up to 8 characters and is used to identify the program on the cassette. BASIC responds with:

```
Cassette ready (Y/N)
```

The tape recorder should be connected and set to play. Type "Y" in answer to the question. If the remote start socket on the tape recorder is connected, BASIC will start the tape and search it for the specified filename. Any other programs found will be reported:

```
Found "program"
```

until the specified filename is found. When the load is complete, BASIC will print "*Ready". If the program was SAVEd with auto run, the program will run as soon as the load is complete.

A checksum error during loading will produce the message:

```
** Tape read error **
```

and the load is aborted.

The LOAD command with no filename will list the files on the tape. No program is loaded.

To get out of the load command, for example if the filename is not found, press the ESCAPE key.

EXAMPLE:

```
LOAD "DEMO"
Cassette ready? (Y/N) Y
Found "PROG1"
Found "DEMO"

*Ready
```

4.1.4 SAVE

Form:

SAVE "<filename>"

The SAVE command is used to store programs on audio cassette tape. The filename is up to eight characters and is used to identify the program on the tape. BASIC responds with:

Auto-Run ? (Y /N)

For a normal save of a program type "N". The program can then be reloaded using the LOAD command. If "Y" is used, the auto run flag is set in the header block for the program. This means that when it is reloaded using LOAD, it will run as soon as the load is complete without having to type RUN.

The next prompt is:

Cassette ready? (Y /N)

The tape recorder should be connected and set to record. If the remote socket on the tape recorder is connected, BASIC will turn on the tape and save the program. When the operation on complete, the "*Ready" message w111 be printed.

EXAMPLE:

```
SAVE "DEMO" •
Auto-Run ? (Y/N) N
Cassette ready? (Y/N) Y
*Ready
```

Cassette data is dumped as a memory image at 300 baud. The format is shown in the table.

TAPE SAVE FORMAT

SYNC CHAR (16H)	Repeated for two second startup
STX CHAR (02H)	Start of data
HEADER BLOCK	See below
MEMORY IMAGE	
ETX CHAR (03H)	End of data
CHECKSUM	Checksum of memory image

HEADER BLOCK

AUTO RUN FLAG	RUN =>A5A5 NORUN =>5A5A
8 BYTE NAME	Null filled when name is less than 8 characters long
POINTER 1	The pointers are used to initialize the BASIC program when it is loaded
POINTER 2	
POINTER 3	
LOAD LENGTH	(BYTES)
CHECKSUM	Header only

4.1.5 LIST

Forms:

```
LIST
LIST <line number>
LIST <line number> TO <line number>
LIST TO <line number>
```

The LIST command displays all or any portion of the current program. Entering only LIST lists the entire program.

Entering a line number displays just that line. Entering two numbers separated by TO displays all lines between and including those specified. Omitting the first line number in this form displays from the start of the program to the specified line number.

A listing may be halted by pressing the space-bar. Successively pressing the space bar will step the listing one line at a time. Pressing RETURN resumes the listing.

To end a listing before it is complete press ESCAPE.

A listing may be sent to other devices by using the UNIT statement. The listing will always proceed at the speed of the slowest device enabled.

EXAMPLES:

```
LIST 100 TO 3000
LIST TO 500
```

4.1.6 RUN

Form:

RUN

The RUN command clears all variable space, pointers and stacks. The random number seed is set to 0 and then the current BASIC program is executed from the lowest line number.

4.1.7 CONT

Form:

CONTinue

The CONTINUE command transfers control to the next statement of the BASIC program. (The RUN command always starts at the first line.)

When the RUN command is entered, program execution begins at the first line and continues until a break condition occurs. The CONTINUE command may be used to continue execution after a break.

The program will stop or break when the user enters the ESCape key during program execution, a STOP or END statement is encountered, or an error occurs within the program.

If execution was halted by an error or the 'escape' key, then the interrupted line will be re-executed by "CONTINUE". If execution was halted by a "STOP" statement, "CONTINUE" will execute the following line. It is not possible to "CONTINUE" past an "END" statement.

4.1.8 NUMBER

Forms:

```
NUMBER
NUMBER <line number>
NUMBER <line number>,<increment>
```

The NUMBER command allows entry of BASIC programs without having to type in the line numbers. The user is prompted with the next line number each time a line is completed. Entering just the command will start lines at 100 with an increment of 10. The second form allows the user to choose the start line number and the third form allows the user to choose the increment between lines.

Each line is terminated by RETURN, producing the next line number. If RETURN is pressed without entering a statement, the NUMBER mode is terminated.

EXAMPLE:

```
NUMBER 200,20
200 <enter your line here - end with RETURN>
220 <and so on>
...
...
380 <press RETURN ONLY to finish entering>
```

4.1.9 RENUM

Form:

```
RENUM [new start line number] STEP [new increment]
```

The RENUM command renumbers the current BASIC program. On completion, the first line is changed to that specified with the difference between adjacent lines equal to the new increment. All references to line numbers within the program are changed so that they refer to the new line numbers (e.g., GOTOs, GOSUBs).

If the increment is omitted, 10 is used. If the new start line number is omitted, 10 is also used for this.

EXAMPLE:

Given the program:

```
10 REM THIS IS A DEMO PROGRAM
17 PRINT "TESTING"
21 GOTO 17

RENUM 100 STEP 20

LIST

100 REM THIS IS A DEMO PROGRAM
120 PRINT "TESTING"
140 GOTO 120
```

The command is implemented using a two-pass approach. On the first pass the command determines how much free memory exists in the system and where it is. This area of memory is then used to build a table of line number references. The whole program is scanned and every time a line number is encountered within the program an entry is added to this table. Each table entry consists of the line number and the address in program memory where it was found. If there is insufficient free memory to complete this table then the table area is reset to its original state, the error message '**Out of memory**' is output and the RENUMber command is abandoned.

However, if the first pass is completed successfully, (there was sufficient memory) then, and only then, is the second pass executed. This approach ensures the integrity of the stored program. It cannot get corrupted due to starting to renumber the program and not being able to complete the operation.

If there are any line numbers in the form of expressions (e.g., ERROR A), this is detected by the first pass. It is not possible to update these references automatically so the following warning message is output:

```
Problem with new line xx
```

where "xx" is the line containing the unconverted reference.

When the renumbering is complete, check the line(s) indicated and convert the statement references by hand. All other references will be converted correctly.

In the second pass each line number entry in the table is checked to determine whether or not the line it refers to exists. If it does, that line's new value is used to update the memory locations that refer to the line. When a line that does not exist is referred to then the following message is output:

```
Bad Line No.(xx) in new line yy
```

where "xx" is the line that does not exist.
"yy" is the line with the reference to "xx".

When all table entries have been completed the free memory area is put back to its original state and the Cortex is ready for more commands.

If the SIZE command is executed immediately before and then straight after executing the RENUM command, there may be a slight difference between the two program sizes shown. This is nothing to worry about; it is the result of internal reorganization and does not affect the operation of the stored program.

If the program to be renumbered is very large or requires a vast amount of data storage (for instance, very large arrays) then it is possible that the command will fail due to the limited amount of temporary storage space available.
This may be increased by entering line 1 as STOP. Then type RUN. This will clear the arrays. If there is still insufficient space, the following will clear all variables:

- SAVE the program.
- Execute the NEW command.
- LOAD the program.
- Re-issue the RENUM command.

For an extremely large program, it will have to be saved on tape in parts using source format and renumbered in parts. References between these parts must be converted by hand.

4.1.10 PURGE

Form:

PURGE <start line number> TO <end line number>

The PURGE command deletes lines from the current BASIC program from the start line to the end line inclusive.

4.1.11 MON

Form:

MON

The MON command executes the assembly language/machine code monitor. Section 7 is the reference guide for the monitor. Once in the monitor, the prompt "[]" is displayed. A "G" command will return to BASIC.

4.1.12 BOOT

Form:

```
BOOT
```

NOTE: Do not use the BOOT command if you do not have floppy discs.

The BOOT command loads the contents of track zero sector one from floppy disc drive number zero and executes this as a TMS9995 program. This program would normally be a secondary boot for the software on the disc.

The disc format is compatible with IBM 3740 for single density and IBM system 34 for double density.

TRACK ZERO HEADER FORMAT

Address	0	2	4	6	8	A	C	E
00	WP	PC	---	---	---	---	---	---
10	---	---	LN	SA	---	---	---	CS

WP = Entry Workspace pointer
PC = Entry Program counter
LN = 256 times number of sectors to transfer
SA = Load address for absolute code program
CS = Checksum on words hex00 to hex1C inclusive

The rest of the words in the header are not interpreted by the BOOT command, but they are loaded into memory and may be used to store other vectors.

The bytes to be transferred are stored sequentially after this header block. This code is first loaded into a buffer that is 1 Kilobyte long. Loading more than this will start to overwrite BASIC, which or may not be desirable depending on the application. The code, including the header block, is then transferred to the load address given by SA.

The maximum amount of code that may be directly loaded by the BOOT command is 1 track. Note that DMA transfers are not possible above address F000 hex.

The LN word in the boot header is 256 times the number of sectors to transfer. For single density discs with 128 bytes per sector (IBM 3740 compatible) this means that LN is twice the number of bytes transferred. For double density discs with 256 bytes per sector (System 34 compatible) LN is exactly the number of bytes transferred. Other values of bytes per sector may be used by adjusting LN accordingly.

The type of disc for booting is determined by the size and density jumpers on the CORTEX board. These should be set as follows:

	IN	OUT
Size	5 inch	8 inch
Density	Single	Double

The bootstrap is always loaded from track zero, which is in the same place for single or double sided discs. Therefore no jumper is needed to select the number of sides.

Recording method is FM for single density and MFM for double density.

For booting, the timing parameters for the discs are set to the slow values shown below. This will allow booting from most drives. After the bootstrap is loaded, it can then access the floppy disc controller to optimize these settings for the drives in use.

Initial timing settings

	5 inch	8 inch
Head step time	50mS	10mS
Head settle time	35mS	8mS
Head load time	--	35mS

4.1.13 ERROR

Form:

```
ERROR
```

When used in a program, ERROR is a statement to trap errors, see section 4.2.5. When entered without a line number, ERROR is a command.

Normally when a line is entered that has errors, Cortex will print the line again and position the cursor for the error to be corrected. The ERROR command turns off this feature. Any lines with errors will be ignored. A subsequent ERROR command turns the error editing back on.

The current state of error editing is returned by the SYS(8) function. A value of 0 means error editing is on, -1 means that it is off.

The ERROR command may be used to load all correct lines in source format from a serial device where errors are occurring. Normally, loading would stop at the first error.

4.2 BASIC STATEMENTS

4.2.1 REM

Form:

```
<line number> REM <text>
```

The REM statement is used to insert remarks (comments) in a program. REM may contain any textual information. It has no effect when encountered in execution; however, its line number may be used as the argument of a GOTO or GOSUB statement. Tail remarks may also be inserted into a program by separating the remark field from the statement field by an exclamation point (!).

Examples:

```
10  REM THIS IS A COMMENT
100 REM CHECK FOR X=0
```

4.2.2 DIM

Dimension declarations are used to specify the size attributes for subscripted variables within the program.

Form:

```
<line number>  DIM <var(dim[,dim]...) [,....]>
               DIM <var(dim[,dim]...) [,....]>
```

The DIM statement dynamically allocates user variable space for array variables. Dimensioned (array) variables must be declared by the DIM statement before the variables are used. Once dimensioned, attempts to redimension an array variable to a larger array size will result in an error message, and attempts to redimension to a smaller size will be disregarded.

Array sizes are specified by indicating the maximum subscript values in parentheses following the array name. Subscripts of dimensioned variables may be any numeric quantity including constants, simple variables, other dimensioned variables, or even function calls. If a floating point value 1s returned for the subscript value, only the integer portion will be used in the dimension statement. The number of dimensions and the dimension size for the array declaration is limited only by the user’s available memory. An error will occur if the dimensioned variable requires more variable space than is currently available in the user's partition. Dimensioned variables always use the 0 subscript as the first element in the array.

Examples:

```
10   DIM A(10),B(10,20)
100  DIM A1(10),B1(20,30),B15(10,10,10)
     DIM CAT(C,D),DOG(SQR(N),3,P)
```

The first statement allows for the entry of an array of 11 elements (0-10) into A, and of an array of 11 x 21 elements into the two dimensional array, B. The two remaining statements dimension arrays in a similar manner.

String variables are also dimensioned using the DIM statement. The "$" sign is used to signify that the variable is being used to store a string. A "$" is not required in the DIM statement but it is useful to include it for your own reference.

Examples:

```
20 DIM $CAT(l0),$DOG(8)
```

Strings are stored one character per byte with a null character used to terminate the string. Hence, simple string variables and single array elements (which are 6 bytes in length) can contain up to five characters. String arrays can contain up to the number of elements times 6 minus 1 character. Therefore, the string array $CAT can contain up to 65 characters. Section 6 gives more information about character strings.

4.2.3 DEF

The DEF statement defines a user function. The defined functions are executed only when the function is referenced.

Forms:

```
<line number> DEF FN <letter> = <expression>

<line number> DEF FN <letter> (parm1[,parm2][,parm3]) = <expression>
```

where:

parameters are single alphabetic letter dummy variables
expression is any valid POWER BASIC expression

The DEF statement may appear anywhere within a BASIC program and the defined functions may be used in any expression. That is, once defined, the functions may be used in the same way as the built-in mathematical functions explained in Section 7. When the function is referenced, the expression is evaluated end the parameters, if any, are replaced by the arguments given in the reference. Within the expression the parameters may appear only as numeric variables. The user may define functions using up to three dummy parameters. All (dummy) parameters may only be single character variables in the function definition. However, when calling the function the user may use any valid POWER BASIC variable (either simple or dimensioned) to replace the dummy variables of the called function.

The expression may include any combination of intricate functions, other user-defined functions, or may involve any other variables in addition to the ones used in the argument of the calling function. Parameter names are dummy (local) variables of the defined function, and have no meaning outside of the function definition.

The use of the DEF statement is limited to those functions whose expression may be evaluated within a single BASIC statement.

The name of the defined function must be three letters, the first two of which must be FN followed by a single letter; e.g., functions FNA through FNZ may be defined by the user. The same letter which defined the function may also be used as a parameter of the function as shown below.

Example:

```
20 DEF FNA(X,Y)=X/Y+5
30 DEF FNB = A/B + C-15
40 DEF FNC(I,J) = I*K/J + FNB - FNA(I,J)
50 DEF FND(N) = N*N/2
60 DEF FNI(I,J) = I*J/SQR(I)
```

4.2.4 LET

The LET statement assigns a value to a variable where the variable is set equal to an expression consisting of variables and/or constants separated by operators. The variable being evaluated may appear within the expression. The newly calculated value of the variable replaces the old value.

In POWER BASIC the letters LET may be omitted from the statement so only an equation appears. The LET statement may have either of the following forms:

```
<line number>  LET <variable> = <expression>
               LET <variable> = <expression>
<line number>  <variable> = <expression>
               <variable> = <expression>
```

where

variable is a string variable, numeric scalar variable, or array element.

The assignment statement assigns an expression value to a variable. Both variable and the expression must be either string or numeric. The following examples illustrate the assignment statement. Note that this is not a meaningful POWER BASIC program.

```
      A=5
      B=10
      LET C=A+B
10    LET X=1
20    LET $A(2)=$C+"NOW"
30    LET Q2(L)=Q2(L+1)+3
40    LET H=6
50    D=5
60    F=A/B+3
100   LET Z[I,J]= 3*X-4*Y
120   $AB="STOP"
```

4.2.5 Control Statements

4.2.5.1 GOTO

When the computer encounters a GOTO statement, it jumps to the program line number specified in the statement. The program executes the statement at the specified line number and continues in sequence with the statements that follow.

Form:

```
<line number>  GOTO <line number>
               GOTO <line number>
```

The "GOTO" statement must be entered without any embedded blanks. If the GOTO statement is not preceded by a line number, program execution begins at the line number specified immediately after the GOTO statement.

Examples:

```
      GOTO 200 Begins execution at statement 200
100   GOTO 140 Transfers control to statement 140
```

The following program illustrates the GOTO Statement:

```
20 INPUT A
30 GOTO 50
40 STOP
50 PRINT A
60 GOTO 40
```

The program execution sequence is line numbers 20, 30, 50, 60 and 40 where execution stops.

4.2.5.2 IF-THEN

IF-THEN statement. The IF statement alters sequential execution of the program depending on the state of the specified condition.

Forms:

```
<ln> IF <expression> THEN <BASIC statement(s)>
     IF <expression> THEN <BASIC statement(s)>
<ln> IF <expression><relation><expression> THEN <BASIC statement(s)>
     IF <expression><relation><expression> THEN <BASIC statement(s)>
<ln> IF <string><relation><string> THEN <BASIC statement(s)>
     IF <string><relation><string> THEN <BASIC statement(s)>
<ln> IF <string> THEN <BASIC statement(s)>
     IF <string> THEN <BASIC statement(s)>
<ln> IF <string><relation><string><,expression> THEN <BASIC statement(s)>
     IF <string><relation><string><,expression> THEN <BASIC statement(s)>
```

The condition may be any variable, numeric expression, relational expression, logical expression, string variable, string relational expression, or function which can evaluate to a zero or non-zero value. Two expressions or strings are compared according to the given relation and a true or false condition results. If the second string is followed by a comma, the expression following the comma indicates the number of characters to be compared. If only a single expression or string is given, the condition is considered false if the expression is zero or the string is null; otherwise, it is considered true.

If the condition is true, the statement(s) following the THEN clause on the same line will be executed. If the condition is false, the statement on the line following the IF-THEN statement will be the next statement executed. Any POWER BASIC statement or statements (including GOTO's and other IF-THEN statements) may immediately follow the THEN clause. They cannot extend to the next statement line because statement execution continues at the next statement line when a false condition occurs. The IF and THEN clauses must appear on the same statement line.

Examples:

```
20 IF A=0 THEN GOTO 100
30 IF SQR(J) =4 THEN K=J*J/I::PRINT J,K
AO IF I+2 THEN PRINT I
50 IF $A=$B THEN PRINT $A
60 IF $A THEN $B=$A
70 IF $A=$B,3 THEN GOTO 200 (compares first three
                              characters of $A and $B)
```

4.2.5.3 ELSE

ELSE statement. The ELSE statement enables conditional execution of POWER BASIC statements depending upon the true or false condition of the last executed IF statement.

Form:

```
      <line number) ELSE <BASIC statement)
```

IF-THEN statements set the ELSE flag to indicate the true or false condition of the last executed IF-THEN statement. Subsequent ELSE statements use the ELSE flag to determine whether the statement(s) following the ELSE are to be executed. When the IF condition is true, the THEN clause will be executed and all subsequent ELSE statements will not be executed. When the IF condition is false, the THEN clause will not be executed and all subsequent ELSE statements will be executed. The ELSE statement must not be placed on the same statement line as the preceding IF-THEN statement because when the IF condition is false, no further statements on the IF-THEN line will be executed and execution will continue with the next statement line. The ELSE flag remains set to the true or false condition until the next IF-THEN statement is executed at which time the flag is cleared and set to the new true or false condition. Several ELSE statements may appear between each IF-THEN statement, and each of these will be executed between each IF-THEN statement; they will be executed when they are encountered if the last executed IF-THEN statement resulted in a false condition. If a true condition resulted, each of these statements will be skipped. An ELSE statement always uses the last IF statement executed as its reference regardless of where it physically lies within the POWER BASIC Program. This enables blocks of statements to be conditionally executed or skipped.

Example:

The following program computes the function and prints the result:

Statement of function:

```
      for X<1, f=ABS(X),
      for 1< =X<2, f=SQR(X),
      for 2< =X, f=ABS(X)-SQR(X)
```

Program solution:

```
      10 IF X<1 THEN F=ABS(X)
      20 ELSE IF X<2 THEN F=SQR(X)
      30 ELSE F=ABS(X)-SQR(X)
      60 PRINT X,F
```

4.2.5.4 GOSUB, POP, RETURN

BASIC programs may contain internal BASIC subroutines. An internal subroutine is a sequence of BASIC statements performing a well-defined function or operation within the POWER BASIC program. Three types of statements govern access to a subroutine: a GOSUB statement for entry into the subroutine, a POP statement for exiting nested subroutines, and a RETURN statement for return to the calling program.

Forms:

```
      <line number> GOSUB <line number>
      <line number> POP
      <line number> RETURN
```

An internal POWER BASIC subroutine may be invoked from any point within the program by using a G0SUB statement which specifies the entry point of the subroutine as a line number.

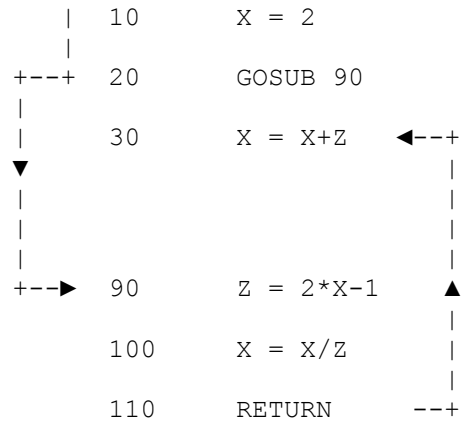


FIGURE 5-1. GOSUB Example

Execution of the GOSUB statement pushes the address of the statement immediately following the GOSUB statement onto the GOSUB stack for return, and passes execution to the specified line number.

A RETURN statement placed in the subroutine is an exit point from the internal POWER BASIC subroutine. A RETURN statement should be placed at each logical end of all subroutines. The RETURN statement causes execution to resume at the first statement following the GOSUB statement that transferred to the subroutine. During this transfer, the top return address is removed from the GOSUB stack. All subroutines should be exited only via a RETURN statement so the top return address will always be removed from the GOSUB stack. Unpredictable results occur if a subroutine is exited in any other fashion.

In Figure 5-1 GOSUB 90 involves statements on line numbers 90 (start of subroutine), 100, and 110 (end of subroutine). If a GOSUB statement is used, the subroutine it branches to must contain at least one RETURN statement. The example illustrates the simplest use of GOSUB and RETURN. The arrows indicate the flow of control in the program.

Subroutines may be nested by a subroutine containing a call to another subroutine (the inner subroutine is called a nested subroutine).

Subroutines may be nested up to 20 levels.

A return address (first line number after the call) must be stored for each GOSUB statement until that statement is executed. The program in the following example contains nested subroutines and shows the actual execution sequence. Each GOSUB to a subroutine must be accompanied by at least one RETURN statement per exit path. The nested program and execution sequence of the example demonstrate entry to and exit from a subroutine.

```
LIST

      10 PRINT "ROOTS OF QUADRATIC EQUATIONS"
      20 PRINT
      30 REM - ENTER COEFFICIENTS A,B,C OF A*X*X+B*X+C
      40 INPUT "COEFFICIENTS A= ";A;"   B= ";B;"  C= ";C
      50 GOSUB 100
      60 REM - RESTART OR END PROGRAM?
      70 INPUT "MORE DATA (1=YES, 0=NO)? "%1;N
      80 IF N<>0 THEN GOTO 20
      90 STOP

      100 REM - CALCULATE S=B*B-4*A*C
      110 S=B^2-4*A*C
      120 REM - COMPLEX ROOTS?
      130 IF S<0 THEN GOSUB 200   !COMPLEX ROOTS
      140 ELSE GOSUB 300          !REAL ROOTS
      150 PRINT                   !OUTPUT BLANK LINE
      160 RETURN

      200 REM - CALCULATE COMPLEX ROOTS
      210 Q=SQR(ABS(S))
      220 R1=-B/(2*A)             !REAL PART
      230 R2=Q/(2*A)              !IMAGINARY PART
      240 PRINT "ROOTS (COMPLEX):  ";R1;" + OR -";R2;" I"
      250 RETURN

      300 REM - CALCULATE REAL ROOTS
      310 IF S=0 THEN Q=0
      320 ELSE Q=SQR(S)
      330 R1=(-B-Q)/(2*A)         !ROOT 1
      340 R2=(-B+Q)/(2*A)         !ROOT 2
      350 PRINT "ROOTS (REAL):  ";R1;",  ";R2
      360 RETURN
```

would Produce the following results:

```
RUN

ROOTS OF QUADRATIC EQUATIONS

COEFFICIENTS A= 2 B= 1 C= -1
ROOTS (REAL):    -1,    0.5

MORE DATA (1=YES, 0=NO)? 1

COEFFICIENTS A= 1 B= 4 C= 6
ROOTS (COMPLEX): -2 + OR - 1.414214 I

MORE DATA (1=YES, 0=N0)? 0

STOP AT 90
```

The following example shows the execution sequence of the previous example. Note that all returns are performed via RETURN statements.

Execution sequence:

```
 10 PRINT "ROOTS OF QUADRATIC EQUATIONS"
 20 PRINT
 30 REM - ENTER COEFFICIENTS A,B,C OF A*X*X+B*X+C
 40 INPUT "COEFFICIENTS A= ";A;"  B= ";B;"  C= ";C
 50 GOSUB 100

100 REM - CALCULATE S= B*B-4*A*C
110 S= B^2-4*A*C
120 REM - COMPLEX ROOTS?
130 IF S<0 THEN GOSUB 200   !COMPLEX ROOTS
140 ELSE GOSUB 300          !REAL ROOTS

300 REM - CALCULATE REAL ROOTS
310 IF S=0 THEM Q=0
320 ELSE Q=SQR(S)
330 R1= (-B-Q)/(2*A)        !ROOT 1
340 R2= (-B+Q)/(2*A)        !ROOT 2
350 PRINT "ROOTS (REAL): ";R1;" , ";R2
360 RETURN

150 PRINT                   !OUTPUT BLANK LINE
160 RETURN

60  REM - RESTART OR END PROGRAM?
70  INPUT "MORE DATA (1=YES, 0=NO)?  "%1;N
8O  IF N<>0 THEN G0TO 20
20  PRINT
30  REM - ENTER COEFFICIENTS A,B,C OF A*X*X+B*X+C
40  INPUT "COEFFICIENTS A= ";A;"  B= ";B;"  C= ";C
50  GOSUB 100

100 REM - CALCULATE S= B*B - 4*A*C
110 S=B^2-4*A*C
120 REM - COMPLEX ROOTS?
130 IF S<0 THEN GOSUB 200   !COMPLEX ROOTS

200 REM - CALCULATE COMPLEX ROOTS
210 Q= SQR(ABS(S))
220 R1= -B/(2*A)            !REAL PART
230 R2= Q/(2*A)             !IMAGINARY PART
240 PRINT "ROOTS (COMPLEX): ";R1;" + OR -"R2;" I"

150 PRINT                   !OUTPUT BLANK LINE
160 RETURN

 60 REM - RESTART OR END PROGRAM?
 70 INPUT "MORE DATA (1=YES, 0=NO)? "%1;N
 80 IF N<>0 THEN 00TO 20
 90 STOP
```

A RETURN statement must not be encountered unless a GOSUB statement has been executed.

"Remembering" all the return points by saving them on the GOSUB stack and never removing them can exhaust the available GOSUB stack area. The following program, which calculates N! illustrates this problem; its use requires that N return points be remembered.

```
10 INPUT "N= ";N
20 GOSUB 100
30 PRINT N,N1
40 STOP

100 N3=N
110 N2=0
120 N1=1
130 GOTO 160
140 N3=N3-1
150 GOSUB 160
160 IF N3>1 THEN GOTO 140
170 N2=N2+1
1E0 N1=N1*N2
190 RETURN
```

The POP statement removes the top most previous return address from the GOSUB stack. It does not perform a return transfer to the calling routines. Execution continues at the statement following the POP statement in the internal subroutine. The POP statement is useful for exiting nested subroutines as the following example demonstrates.

```
 10 REM - MAIN PROGRAM
 20 GOSUB 100  ! CALL GET DATA
 30 . . . . .
       "
       "
       "
       "
       "
100 REM -  ! SUBROUTINE GET DATA
110 GOSUB 200  ! CALL GET NUMBER
120 . . . . .
       "
       "
       "
       "
       "
190 GOTO 100  ! GET NEXT DATA SEQUENCE
200 REM - SUBROUTINE GET NUMBER
210 . . . . .
       "
       "
       "
       "
       "
250 REM - NUMBER FOUND?
260 IF NUM THEN RETURN  ! IF NUMBER - RETURN
270 REM - NO MORE NUMBERS
280 POP ! REMOVE MOST RECENT RETURN ADDRESS
290 RETURN
```

In this example, the main program calls subroutine 100 which in turn calls subroutine 200 until there is no more data. Subroutine 200 exits with a RETURN when data is found; a POP then RETURN when there is no more data. Program execution then continues at line 30.

4.2.5.5 ON

```
                    <variable>          GOTO
<line number> ON {      or      } THEN {  or   } <line number>,<line number>,..
                    <expression>        GOSUB

                    <variable>          GOTO
              ON {      or      } THEN {  or   } <line number>,<line number>,..
                    <expression>        GOSUB
```

ON statements select the target transfer line number of a GOTO or a GOSUB from a list of statement numbers. The statement number list contains a statement number for each expected value of the expression or variable. The selection is based on the value of the expression or variable truncated to an integer. If the expression value is 1, the first line number in the list is selected. If the value is 2, the second will be executed, and so forth. The GOTO or GOSUB statement will be executed, transferring control to that line. If the expression value is less than one or greater than the number of statement numbers in the list, the transfer is not made and execution simply continues with the next statement.

Examples:

```
10 ON J+1 THEN GOTO 15, 20, 35, 46, 70
```

When J is equal to 3, J+1 is equal to 4, and the fourth statement number (46) is executed next. Similarly, J values of 0, 1, 2, and 4 result in jumps to statement numbers 15, 20, 35, and 70, respectively.

```
110 ON X+3 THEN GOSUB 20, 40, 80, 300
120 ON (A+5)/Z THEN GOTO 10, 30
```

When X is equal to -1, the second statement number (40) is executed next. When X is less than -2 or greater than +1, a transfer is not made and line 120 will be the next statement executed. When (A+5)/Z is equal to 2, the second statement number (30) is executed next and so forth. If the expression evaluates to a non-integer value, only the integer part is used to determine the appropriate branch point.

4.2.5.6 FOR/NEXT

FOR and NEXT statements indicate the start and end of an instruction block that is to be repeatedly executed as a set. One variable takes on different values within a specified range; this variable is often used in the computation or evaluation contained in the instruction block. The FOR statement names the variable and stepping values of that variable and also specifies its initial and final values. The NEXT statement closes the program loop.

The FOR statement may have either of the following forms:

```
<ln> FOR <variable> = <expression> TO <expression>
     FOR <variable> = <expression> TO <expression>
<ln> FOR <variable> = <expression> TO <expression> STEP <expression>
     FOR <variable> = <expression> TO <expression> STEP <expression>
```

where

variable is a simple numeric scalar variable
expression is a valid POWER BASIC numeric expression

The NEXT statement has the form:

```
<line number) NEXT <variable)
              NEXT <variable)
```

where

variable is a simple numeric variable

The simple variable of the NEXT statement must be the same as the FOR statement variable at the beginning of the loop.

Specification of the STEP value is optional and usually omitted. If omitted, a value of +1 is used. The step value may be any constant, variable, or expression which evaluates to a positive or negative value. Negative step intervals can be used to decrease the value of the FOR variable from one pass through the loop to the next. By using a step value of -1, the FOR variable can be made to decrease by integer values during successive loop interactions.

Examples:

```
100 FOR X=0 TO 3 STEP D
200 NEXT X
300 FOR X4=(17+COS(Z))/3 TO 3*SQR(10) STEP 1/4
400 NEXT X4
500 FOR X=8 TO 3 STEP -1
600 FOR J=-3 TO 12 STEP 2
700 NEXT J
800 NEXT X
```

Note that the step size may be a variable (D), an expression (1/4), a negative number (-1), or a positive number (2). In the example with lines 300 and 400, successive values of X4 will be .25 apart in increasing order. In the next example, the successive values of X iterations through the loop, J will take on values -3, -1, 1, 3, 5, 7, 9, and 11.

IF expressions are used to specify the initial, final or step-size values, they will be evaluated only once when the FOR loop is entered. Changing any of the values (either the step, initial or final values) within the FOR loop does not affect the number of times the sequence is executed with the exception of the control variable. The control variable is assigned to the initial values when the FOR statement is entered and is incremented (if the STEP value is positive), or decremented (if the STEP value is negative) after each repetition of the loop sequence. The last repetition of the loop sequence is when the control variable is equal to the final value. When exiting the loop in this manner, the control variable is incremented (or decremented) one step value beyond the final value.

A pre-check is performed so that if the initial value is greater than the final value in the case of positive STEP values, the loop sequence will not be executed. Likewise, if the initial value is less than the final value and the STEP value is negative, the loop sequence will not be executed.

The control variable may be changed within the body of the loop and the latest value of the variable will be used in the exit test;

however, this programming practice is not recommended.

The statement "50 FOR I=2 TO -1" without a negative step size results in the body of the loop not being executed, and execution proceeds to the statement immediately following the corresponding NEXT statement. The NEXT statement must be the first item in a line for this feature to work properly.

The loop continues to be executed as long as the condition:

```
(step value)*(control variable)<(step value)*(end value)
```

remains true. If the condition:

```
(step value)*(start value)>(step value)*(end value)
```

is true when the FOR statement is first encountered, the loop will not be executed.

When the loop is being executed, the control variable is first set to the initial value and if the end criterion is not true, the loop is executed. The control variable is then incremented by the step value each time the NEXT statement is encountered and executed. The loop terminates with the control variable equal to the last value used in the loop plus the step value.

Example:

```
10 FOR I=1 TO 4 STEP 2
 .
 .
 .
80 NEXT I
90 PRINT "I=";I

RUN
I= 5
```

The NEXT statement closes the FOR loop. When it is encountered, the step value is added to the control variable. If the control variable has not gone beyond the end value, control will be returned to the first statement following the FOR which opened the loop. The control variable of the loop to be closed must be specified by the NEXT statement. It is possible to place the FOR and NEXT statements on the Basic statement line; however, remember that statement lines are autonomous. Therefore, this type of loop structure cannot be interrupted by using the escape key since keyboard sampling is performed only between statement lines.

Also, FOR/NEXT statements on a single line or in separate statement lines will cause an error to result if, during the initial pre-check,

the initial value has exceeded the final value. For example,

```
20 FOR I=10 TO 1::NEXT I
```

will result in an error

FOR loops may be nested; i.e., one FOR loop may contain another which may contain a third, etc. If nested, however, they should not use the same control variable. When two loops are nested, one must be completely contained within the other. Overlapping is not permitted. The following structure is correct:

```
 10 FOR I=1 TO 2
 20 FOR J=1 TO 2
 30 FOR K=1 TO 2
      .
      .
      .
 80 NEXT K
 90 NEXT J
100 NEXT I
```

while the next two structures are incorrect:

```
 10 FOR I=1 TO 2
 20 FOR J=1 TO 2
      .
      .
 80 NEXT I
 90 NEXT J (WRONG, loops may not overlap)

 10 FOR I=1 TO 2
 20 FOR I=1 TO 2
      .
      .
 80 NEXT I
 90 NEXT I (WRONG, nested loops may not have the same
            control variable.)
```

The following program illustrates nesting:

```
LIST
   10 REM AREA OF A TRIANGLE
   20 FOR B=6 TO 9
   30 FOR H=11 TO 13 STEP 0.5
   40 A=B*H/2
   50 PRINT B,H,A
```

```
60 NEXT H
70 NEXT B
80 STOP
```

This program prints the base, height, and area of triangles with bases 6, 7, 8 and 9, and heights 11, 11.5, 12, 12.5, and 13. All combinations are printed; 20 sets of data for the four bases and five height values.

All values of the variable in the inner loop are cycled through while the variable in the outer loop is set to its first value. The outer loop variable is then set to its second value and the inner loop is cycled through again. The program runs through each outer loop value this way.

Nesting of FOR/NEXT loops is permitted to a level of 10.

It is legal to transfer control from within a loop to a statement outside the loop, but it is never advisable to transfer control into a loop from outside. The next two examples illustrate both of these situations.

Valid transfer out of a loop:

```
20 FOR I=l TO N
30 X=X=2*I
60 IF X>1000 THEN GOTO 100
50 NEXT I
```

Invalid transfer into a loop:

```
20 GOTO 50
30 FOR I=l TO N
40 X=X*2*I
50 Y=Y+X/2
60 NEXT I
     .
     .  (WRONG, 50 is inside a loop)
     .
```

However, it is permissible to call a subroutine from within a loop and then return from the subroutine back into the loop. The following example illustrates repetitive calling of a subroutine from inside a loop.

Example:

```
 10 FOR I=1 TO N
 20 X=2*I-1
 30 GOSUB 150
 40 Z=Z+Y
 50 NEXT I
      .
      .
      .
150 IF X<>12 THEN G0TO 180
100 Y=248
170 RETURN
180 Y=200+4*X
190 RETURN
```

4.2.5.7 ERROR

The ERROR statement specifies a subroutine that will be called via a GOSUB whenever any POWER BASIC error occurs.

Form:

```
<line number> ERROR <line number>
              ERROR <line number>
```

The ERROR statement enables the user to trap to an internal error processing routine on the occurrence of any error. When an ERROR statement has been executed and an error occurs, control passes to the specified line number via a GOSUB statement. The statement number where the error occurred will be placed on top of the GOSUB stack; if the error is recoverable, a RETURN statement will resume execution at that same statement when the error is corrected. If the error is unrecoverable and control will not be transferred back via a RETURN, it is good programming practice to execute a POP statement to remove the line number from the top of the stack. This practice avoids unnecessary cluttering of the stack, which may cause unpredictable results. After the error trap, the system function SYS(1) will contain the error code number and SYS(2) will contain the statement number in which the error occurred. These are necessary for processing in the error handler subroutine.

Once an error is encountered and causes transfer to the error handler subroutine, the ERROR statement flag is cleared, and future errors will not be trapped unless an ERROR statement is again executed. When an ERROR statement has been executed and an error occurs, the automatic printing of the error code is suppressed.

Example:

```
 100 ERROR 1000
       .
       .
       .
1000 IF SYS(1)=10 THEN PRINT "STORAGE OVERFLOW"::STOP
1010 IF SYS(1)=23 THEN RESTOR::RETURN (rewind data file)
1020 ELSE PRINT "ERROR*" SYS(1):: STOP
1030 RETURN
```

Statement 100 designates the subroutine starting at statement 1000 to be the error handling subroutine. When an error occurs, control is transferred to statement 1000, and the error number is first tested for "storage overflow". If "storage overflow" is not the error, it is tested for the "read out of data" error number. If this is true, the data is restored to its beginning and control returns to the statement in which the error occurred. If this still was not the error, the error number is output and execution stops.

4.2.5.8 STOP

The STOP statement terminates program execution at the logical end of the program. There may be one or more STOP statements in a POWER BASIC program, and they may appear anywhere within the program.

Form:

```
<line number> STOP
```

The system displays the line where program execution terminated.

Example:

```
 900 STOP
STOP AT 900
```

4.2.5.9 END

The END statement marks the end of a program and terminates program execution.

Form:

```
<line number> END
```

The END statement functions just like the STOP statement. It may appear as any statement within the program. The system displays the statement number where program execution terminated.

Example:

```
  70 END
STOP AT 70
```

4.2.6 Internal Input Statements

READ, DATA, and RESTOR statements are used in the following forms:

```
                     <numeric variable>   <numeric variable>
<line number> READ { or } { or }. . .
                     <string variable>    <string variable>

                        <expression>        <expression>
<line number> DATA {<string variable>} {<string variable>}. . .
                    <string constant>    <string constant>

<line number> RESTOR

<line number> RESTOR <line number>
```

POWER BASIC permits definition of a list of data items containing both strings and numbers within the program. Entries in this list are defined by DATA statements and accessed sequentially by READ statements. The RESTOR statement is used to move to a specific point within the list or to the beginning of the list.

4.2.6.1 DATA

The DATA statement contains a list of data items separated by commas. Each item in the list is either a string constant or an expression which evaluates to a numeric constant. String constants must be enclosed in quotes.

Example:

```
      10 DATA 5, 3.14159,"DOE,JOHN",4*ATN(1)
```

A program may contain any number of DATA statements with no restriction on their placement within the program; however, they are typically placed together in a data block near the beginning or end of the program. The data list will contain all of the data items from all DATA statements in the same order they are written in the program. DATA statements have no effect when encountered during execution.

4.2.6.2 READ

The READ statement assigns values from the internal data list to variables or array elements. The first READ statement executed normally starts with the first item in the data list. Reading of data items continues sequentially unless a RESTOR statement is executed. An error is generated when a READ statement requests the next value with the data block exhausted of data.

The READ statement specifies a list of variables or array elements whose values are to be assigned from the data list as shown below:

```
50 READ X, Y, A(5,X), $B,$C(Y)
```

The examples below illustrate use of the DATA and READ statements:

```
10 READ A,B,C,D
20 H=A*B*C*D
30 PRINT A,B,C,D,H
40 READ E,F,G
50 H=E*F*G
60 PRINT E,F,G,H
70 DATA 2,3,5,7,11,13,17
80 STOP

RUN
 2       3       5       7       210
 11      13      17      2431
```

The data in this example is supplied in one DATA statement, but is used in two READ statements at two different locations in the program. When the program encounters the first READ statement, it searches for the lowest-numbered DATA statement (which may occur before or after the READ statement). The program takes numeric values from the DATA statement in sequence associating them with READ statement variables in sequence. In the example, A is assigned the value 2, B the value 3, C the value 5, and D the value 7. The program establishes access to the next data value (11), so it may be assigned to the first variable encountered in the next READ statement. Line 20 is computed, and the newly-introduced variable H is assigned its computed value. The next READ statement at line 40 introduces three new variables. The DATA statement continues to supply data from line 70 at the pre-established access point, so the new variables E, F, and G take on the values 11, 13 and 17. A new value for H is computed in line 50. The statement that follows prints the new values for E, F, G, and H.

The user must match numeric variables in the READ list to numeric expressions in the data list. Similarly, the user must match string variables in the READ list to string constants or string variables in the data list. An error will result if this convention is not followed.

Example:

```
10 READ A,B,$CAT
20 LET C=A+B
30 PRINT A,B,C,$CAT
40 DATA 2,3,"TEXT"
50 STOP

RUN

2       3       5       TEXT
```

4.1.6.3 RESTOR

The RESTOR statement is used to move either to a specific point in the data list, or to the beginning of the list. A RESTOR statement without an argument resets the pointer to the beginning of the first DATA statement.

A RESTOR with an argument resets the pointer to the line number specified. The line number specified must exist but need not be the line number of a DATA statement. The next sequential DATA statement will be used.

Example:

```
70 RESTOR (restores to the beginning of the data list)
80 RESTOR 20 (restores to the first DATA statement at
              or beyond line 20)
```

The following example program illustrates the use of RESTOR:

```
10 DATA 14,16,18
20 READ I,J,K
30 PRINT I,J,K
40 RESTOR
50 READ X,Y,Z
60 PRINT X,Y,Z
70 END

RUN

14      16      18
14      16      18
```

The RESTOR statement in this program resets the DATA pointer and transfers control to the READ statement in line 50 which then obtains data from line 10 (even though the READ statement in line 20 has used the same data). If the RESTOR statement was omitted, POWER BASIC would print an error message indicating a lack of data for the variables in the READ statement at line 50.

If the following statement is added to the example program between lines 40 and 50:

```
45 DATA 2,24,26
```

The statement at line 50 would still cause the values 14, 16, and 18 to be printed. The RESTOR statement at line 40 results in data being obtained from line 10 rather than from line 45.

If a program has no DATA or READ statements, the use of the RESTOR statement does not affect the program.

4.2.7 Input and Output

4.2.7.1 INPUT

The INPUT statement is used for keyboard input into variables of the BASIC program.

Form:

<line number> INPUT <variable> { , ; } <variable> { , ; }

The INPUT statement performs as a READ statement with the exception that it accesses the numeric constants and strings from the external keyboard instead of from internal DATA statements. It provides all translation from character data to the internal formats of the POWER BASIC system and thus assigns input values to the variables or array elements specified in the input list. A11 characters are echoed as they are entered. The INPUT statement is extremely versatile and provides a means to 1) input numbers only, 2) input character strings, 3) detect control characters, 4) prompt with character strings, 5) specify maximum number of input characters, 6) specify exact number of input characters, 7) suppress carriage return/line feed, and 8) suppress prompting.

Input variables may be entered in a list separated by carriage returns. Numeric data may he represented as decimal integers, floating point, exponential, or hexadecimal values. There should be no embedded spaces within numeric values and all spaces preceding or following numeric data are ignored. For string data input, the string consists of all characters after the prompting character and up to (but not including) the end of the input (carriage return). The string includes all entered blanks and quotes.

The INPUT statement prompts the user with a question mark (?) for numeric only inputs, and a colon (:) for character inputs. If an illegal number is entered in response to the question mark prompt, the computer will respond with a double question mark (??) and wait for correct input. The computer will continue to prompt until the user has entered all data requested.

In the following examples, a carriage return is represented by (CR) and all user responses are underlined.

Examples:

```
40 INPUT X
50 INPUT $A, $B
60 INPUT $Y, Z
70 PRINT X, $A, $B, $Y, Z
80 STOP

RUN

 ?256 (cr)
 :CAT (cr) :DOG (cr)
 :HI (cr) ?80A (cr) ??80 (Cr)
 256       CAT       DOG       HI       80

 STOP AT 80
```

In the program, statement 40 outputs a question mark waiting for numeric input. The user enters the number "256" followed by a carriage return which terminates the INPUT statement of line 40. The variable X is assigned the value of "256". Next it prompts with a colon awaiting character string input. The user enters "CAT" followed by a carriage return. The computer immediately prompts with a colon awaiting the next string input. The user enters "DOG" and a carriage return which terminates this input line. The computer then prompts with a colon and the user inputs "HI" and a carriage return. Next, the computer prompts with a question mark and the user incorrectly enters "80A", an illegal numeric value. Therefore, computer responds with a double question mark and awaits correct input. The user enters "80" followed by a carriage return which terminates the INPUT statement. Statement 70 is then executed and outputs the values read into the variables.

An INPUT statement can be combined with a PRINT statement to prompt user response as follows:

```
20 PRINT "YOUR VALUES OF X, Y, AND Z ARE";
30 INPUT X, Y, Z
40 STOP

RUN

 YOUR VALUES OF X, Y, AND Z ARE? 50 (cr) ?60 (cr) ?70 (cr)

 STOP AT 40
```

Since user prompting for data input is required in most applications, the INPUT statement has been designed to permit string constants to be embedded in the INPUT statement for direct prompting output. The string constants must be enclosed by quotation marks. There may be any number of string constants within the INPUT statement separated from input variables and other string constants by commas or semicolons.

The above example may be performed as follows:

```
20 INPUT "YOUR VALUE OF X IS", X, " Y", Y, " Z", Z
30 STOP

RUN

 YOUR VALUE OF X IS? 1 (cr) Y? 2 (cr) Z? 3 (cr)

 STOP AT 30
```

Similarly for string input:

```
10 DIM N(5)
20 INPUT "WHAT IS YOUR NAME", $N 0
30 PRINT "YOUR NAME IS ";$N 0
40 GOTO 20

RUN

 WHAT IS YOUR NAME: JOHN (cr)
 YOUR NAME IS JOHN
 WHAT IS YOUR NAME:
```

A semicolon may be used to perform input formatting. If a semicolon is placed at the end of an INPUT statement line, the carriage return/line feed is suppressed after processing the INPUT statement as the example below illustrates:

```
10 INPUT "INPUT X", X;
20 PRINT " X SQUARED="; X*X
30 INPUT "INPUT Y", Y
```

```
40 PRINT "Y CUBED="; Y*Y*Y
50 STOP

RUN

 INPUT X?12 (cr) X SQUARED= 144
 INPUT Y?3 (cr)
 Y CUBED= 27

 STOP AT 50
```

In line 10 the semicolon is present at the end of the INPOT statement; therefore, the carriage return/line feed is suppressed after entering the constant 12 so that "X SQUARED= 144" can be output on the same line. In line 30 a semicolon is not present so the carriage return/line feed is performed.

When the semicolon is placed before an assignment variable in the INPUT list, the automatic prompting of a question mark or colon is suppressed. The user may then perform his own prompting in the POWER BASIC Program by using PRINT statements or placing character strings in the INPUT statement.

Example:

```
 5 DIM N(3)
10 INPUT "WHAT IS YOUR EMPLOYEE NUMBER?", $N (0)
20 INPUT "WHAT IS YOUR EMPLOYEE NUMBER?"; $N (0)
30 STOP

RUN

 WHAT IS YOUR EMPLOYEE NUMBER?: 1234 (cr)
 WHAT IS YOUR EMPLOYEE NUMBER?1234 (cr)

 STOP AT 30
```

In line 10, the INPUT Statement prompted with a colon (:). In line 20 no prompt was issued.

The user may limit the number of characters which can be entered from the keyboard for both numeric and string variable assignments by using the "#" or "%" operators in the INPUT statement. Use of the "#" operator will specify the maximum number of characters which can be entered from the keyboard. Use of the "%" operator will specify the exact number of characters which must be entered.

Forms:

<line number> INPUT <#> expression {,;} variable {,;}...

<line number> INPUT <%> expression {,;} variable {,;}...

When using the "#" operator, the user may enter any number of characters less than the specified maximum by ending the input sequence with a carriage return. The user cannot enter more than the specified maximum number. When the maximum number of characters has been entered POWER BASIC stops accepting keyboard input, assigns the value just entered, and automatically continues to the next sequential statement or INPUT statement parameter.

Use or the "%" operator requires that an exact number of characters be entered. POWER BASIC waits for the exact number of specified characters to be entered and then continues to the next sequential statement or INPUT statement parameter; no carriage return (cr) is required at the end of user INPUT. If the user attempts to enter less than the specified number of characters by ending the input sequence with a carriage return, POWER BASIC will ignore the carriage return and continue to wait until the number of characters specified has been entered.

Examples:

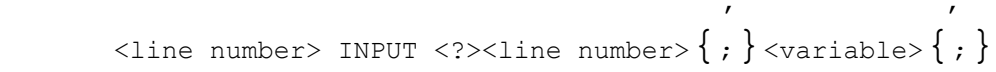
```
10 REM THE MAXIMUM NUMBER WHICH CAN BE ENTERED IS 999
20 INPUT #3, A, B
30 STOP

RUN

 ?512 ?900

 STOP AT 30

10 PRINT "ENTER PHONE NUMBER (XXX-XXX-XXXX)";
20 INPUT %3;A,"-",%3;B,"-",%4;C
30 PRINT "YOUR PHONE NUMBER IS";A;"-";B;"-";C
40 STOP

RUN

 ENTER PHONE NUMBER (XXX-XXX-XXXX)123-456-1234
 YOUR PHONE NUMBER IS 123-456-1234

 STOP AT 40
```

In the first example the user may enter any numbers which do not require more than three keystrokes. The range would be

limited to -99 to 999. In the second example the user is requested to enter his telephone number in the format XXX-XXX-XXXX. The % symbols require the user to enter exactly the required amount of numbers. The user enters 123. The computer places the number in variable A and outputs a "-". The user enters 456, and the computer places the number in variable B and outputs a "-". The user enters 1234 to complete the sequence. Statement 30 then prints the user's phone number using the variables of the INPUT list.

The user may detect any invalid input or control characters which are entered during both numeric and string variable assignment by using the question mark (?) operator in the INPUT list.

Form:



The "?" operator specifies the line number to which control is transferred via a GOSUB statement if a control character or invalid input is encountered during input. The SYS(0) function will return the control character encountered. SYS(0) will be equal to -1 if there was an invalid input. Otherwise, SYS(0) will equal the decimal equivalent of the control character. This feature is useful for transferring control to internal subroutines by using the INPUT statement. For example, to the user who requires additional information for the input of data, (control) H can be used to transfer to a routine which outputs a HELP message.

Example:

```
 10 INPUT ? 100,N
 20 PRINT N
      .
      .
100 REM SUBROUTINE TO PROCESS (control) H INPUT
110 PRINT "USER INPUT ASSISTANCE"
      .
      .

RUN

 ? (control) H -
 USER INPUT ASSISTANCE
      .
      .
```

In line 10 if the user enters a numeric value, it will be entered in the variable N; or if the (control) H key is entered, the subroutine at statement 100 will be executed and output the instructions for user input.

4.2.7.2 PRINT

The PRINT statement causes the values of all expressions in the list to be printed on the output terminal. Commas and semicolons are used to separate expressions and provide for print formatting.

Forms:

```
            PRINT             ,             ,         ,
line number { ; } expression { ; } expression { ; } ... { ; }

            PRINT             ,             ,         ,
          { ; } expression { ; } expression { ; } ... { ; }
```

The expression list may contain any numeric variable, numeric expression, string variable, string constant, or any ASCII code which is to be output to the terminal device.

String constants may be printed directly by inserting them in the PRINT statement expression list. String variables are printed by having the variable name preceded with the dollar sign designator. The following example illustrates the output of string constants and string variables.

```
100 DIM N(10)
110 $N(0)= "POWER BASIC."
120 PRINT "THE NAME OF THE LANGUAGE IS ";
130 PRINT $N(0)
140 STOP

RUN

 THE NAME OF THE LANGUAGE IS POWER BASIC.

 STOP AT 140
```

The PRINT statement may be used to directly output ASCII codes to the terminal device. The hexadecimal ASCII code must be enclosed in angle brackets, (e.g., <0A>) and may be placed anywhere with string constants or predefined string variables appearing within the PRINT statement expression list. Only the low order 7 bits of the hexadecimal code will be output to the device. Evaluation BASIC does not support the direct output of ASCII characters.

Example:

```
10 PRINT "GO TO THE NEXT LINE <0A><0D> AND CONTINUE PRINTING!"
```

would generate

```
GOTO THE NEXT LINE
AND CONTNUE PRINTING!
```

To facilitate rapid statement entry in the edit mode, a semicolon (;) may be used in place of the word "PRINT" in any PRINT statement.

```
20 ;X,Y,Z
```

In its simplest form, the expressions in the output list are separated by commas. In this form, an output line is divided into five 15-character print fields starting in columns 1, 16, 31, etc. A comma following an expression in a list is a signal to advance to the next field. Expressions separated by commas are output one expression per print field. This enables output lines to formatted into five left justified columns within the field. Expressions may occupy more than one field, in which case the comma following the expression in the PRINT list advances the print output to the next blank field. Note that when more than five expressions are included in the output list separated by commas, the terminal device should be of the type which buffers the characters and automatically generates a carriage return/line feed when its buffer is full to obtain the correct five column output. If the terminal device does not perform in this manner, output values may be lost at the end of output lines, and the five column output format may be skewed. Printing will continue in as many lines as are required to complete the output list. When the entire outputs list has been printed, a carriage return/line feed is automatically inserted after the last print item. Subsequent printing begins on the next line. For example, the following statements:

```
10 X=7
20 $NAM = "PAUL"
30 PRINT X, X+2, X+4
40 PRINT "GEORGE", "HARRY", $NAM
```

would generate

```
7              9              11
GEORGE         HARRY          PAUL
```

The automatic carriage return/line feed at the end of a PRINT statement may be suppressed by placing a comma at the end of the output list. Subsequent printing will begin in the next field of the same line. For example:

```
10 X = 7
20 $NAM="PAUL"
30 PRINT X, X+2, X+4,
40 PRINT "GEORGE", "HARRY", $NAM
```

would generate

```
7           9           11          GEORGE       HARRY
PAUL
```

Note that most terminals automatically generate a carriage return and line feed as occurs in the following example:

```
10 FOR I=1 TO 14
20 PRINT I,
30 NEXT I
40 STOP

RUN

 1          2          3          4              5
 6          7          8          9             10
11         12         13         14

STOP AT 40
```

More compact printing can be achieved by using semicolons rather than commas as expression separators. When followed by a semicolon, numbers in the output list will print in as many characters as required to print the numbers of the expression plus one blank space added on the left. However, strings in the output list will print in exactly the end of an output list, the last item will print in a short field as just described, and subsequent printing will begin immediately after that field. For example:

```
10 S1=95
20 S2=87
30 S8=92
40 PRINT "SCORES AND NAME:";S1;S2;
50 PRINT S3;" JOE DOE"
```

would generate

```
SCORES AND NAME: 95 87 92 JOE DOE
```

Another example:

```
10 FOR I=1 TO 14
20 PRINT I ;
30 NEXT I
40 STOP

RUN

1 2 3 4 5 6 7 8 9 10 11 12 13 14

STOP AT 40
```

Note that both semicolons and commas may be used to separate expressions in any PRINT statement and that the print position of the next expression will depend on the separator (semicolon or comma) used to delimit the expressions. The following example illustrates the use of both delimiters in a single PRINT statement.

```
10 H=98
20 L=60
30 A=79
40 PRINT "HIGH= ";H,"LOW= ";L,"AV= ";A
```

would generate

```
HIGH= 98     LOW= 60      AV= 79
```

A PRINT statement without an expression list is a valid statement. Execution of this statement results in the output of one blank line, as the example following illustrates.

```
10 PRINT "THERE SHOULD BE TWO BLANK LINES BETWEEN HERE AND"
20 PRINT
30 PRINT
40 PRINT "HERE!"
```

would generate

```
THERE SHOULD BE TWO BLANK LINES BETWEEN HERE AND


HERE!
```

Print formatting. The PRINT statement may be used to specify the exact print format for the output of numeric expressions. The pound sign (#) within a PRINT statement followed by a hexadecimal formatting character or a decimal formatting string provides this capability.

Forms:

```
                                  ,
<line number> PRINT <#><exp>{;} ....
                                  ,
              PRINT <#><exp>{;} ....
                                   ,
<line number> PRINT <#,><exp>{;} ....
                                   ,
              PRINT <#,><exp>{;} ....
                                  ,
<line number> PRINT <#;><exp>{;} ....
                                  ,
              PRINT <#;><exp>{;} ....
                                                          ,
<line number> PRINT <#><string constant><expression>{;} ....
                                                          ,
              PRINT <#><string constant><expression>{;} ....
                                                          ,
<line number> PRINT <#><string variable><expression>{;} ....
                                                          ,
              PRINT <#><string variable><expression>{;} ....
```

The formatting function may appear anywhere within the parameter list of the PRINT statement. The parameters within the PRINT statement are separated by commas or semicolons as explained in the PRINT statement. A separator appearing at the end of the parameter list will force subsequent printing to continue on the same line just as in the PRINT statement.

A format designator (#) followed by a semicolon, comma, or space is used to output hexadecimal values in either byte, word, or free format, respectively. These format specifiers convert to hexadecimal the numeric constant, variable or expression immediately following the specifier. The scope of the hexadecimal format specifier is for the first statement, variable, or expression only and not for the entire line as in the case of print formatting using a string image. Subsequent values will be printed in free format decimal representation.

The "#;" specifier converts the value and outputs the hexadecimal result is a single byte with no preceding or trailing blanks or zeroes and without the "H" character. Only the least significant byte x111 be output for values which require more than one byte for their hexadecimal representation.

The "#," specifier converts the value and outputs the hexadecimal result as a full word (two bytes) with no preceding or trailing blanks or zeroes and without the "H" character. The least significant two bytes will be output for values requiring more than one word for their hexadecimal representation.

The "#" specifier by itself converts the value and outputs the result in hexadecimal free format representation. The hexadecimal result occupies as many digit positions as required to print the number. It is preceded with a zero (0) and followed by the "H" character.

The following examples illustrate hexadecimal output formatting. The user will terminate the entry line with a carriage return. POWER BASIC outputs are designated by underlining.

```
PRINT #;1;"  ";#,1;"  ";  #1   01  0001  01H
PRINT #;31;"  "#,31;"  "  #31;"  ";31 1F  001F  01FH  31
LET A=106
PRINT #;A;"  ";#,A;"  ";#A;"  ";A   6A 006A 06AH 106
```

Numeric decimal formatting is designated within a PRINT statement parameter list by a print format specifier (#) followed by a format constant or string variable. The format string may be either a string constant enclosed in quotes which directly contains the formatting string, or a string variable which has previously been assigned the formatting string.

The format string indicates the final printed image of how the numeric expressions specified within the PRINT statement parameter list are to be output. Fields are reserved for printing numeric data by forming output images of the printed results. Special characters are used within the format string to indicate these results.

Several formatting strings may be interspersed within a single PRINT statement parameter list. Numeric output values use the last defined print format in that statement line for their output. Exit from a PRINT statement line resets the formatting flag with subsequent numeric values printed in free format. That is, the range of print formatting is limited to the print statement line in which it is located. Subsequent PRINT statements each require their own print format specifier (#) and string.

Text to be output my be interspersed within the formatting string so long as it contains none of the special characters used for print formatting.

The special characters used in the formatting string are shown in Table 5-2.

When using print formatting, floating point numeric values are rounded to the number of decimal plates specified by the format string. A formatting error occurs if a numeric value is inconsistent with the specified formatting string or if the integer portion of a value requires more digits than specified by the format string. This is indicated to the user by filling the entire output field with asterisks (*).

The following paragraphs and examples explain the use of formatting characters. In these examples single quotes (') are embedded within the format field so the actual printed results can be shown more clearly. In practice these quotes typically would not be used. The user may execute these examples from the keyboard by entering the example through the final semicolon (;), inclusive, and then terminating the entry line with a carriage return. POWER BASIC will respond with the formatted results output between the quotes.

The "9" and "0" formatting characters are used as digit holders. The period (.) character specifies the decimal point position on output.

```
PRINT #"'99'"  5;' 5'
PRINT #"'999.00'"25.32;' 25.32'
PRINT #"'99.0'" 15.575;'15.6'
PRINT #"'99.0'" 101.25;******
```

The "0" formatting character also forces a zero if a non-significant digit is output at that position.

```
PRINT #"'999.00'"28;' 28.00'
PRINT #"'990.00'".153;'  0.15'
PRINT #"'990.000'".75;'  0.750'
PRINT #"'990.000'" 1047.23 ;*********
PRINT #"'000-00-000'" 3021; '000-03-021'
```

TABLE 5-2. FORMATTING STRING CHARACTERS

CHARACTER	FUNCTION	EXAMPLE
.	Decimal point specifier	PRINT #"999.99"25.32; 25.32
^	Translates to decimal point	PRINT #"999^00"1000; 10.00
,	Suppressed if before significant digit	PRINT #"999,999.99"100; 100.00
9	Digit holder	PRINT #"9999"123; 123
0	Digit holder or forces zero	PRINT #"9990.99".234; 0.234
$	Digit holder & floats $	PRINT #"$$$.99"8; $8.00
S	Digit holder & floats sign	PRINT #"SSS.99"-6; -6.00
E	Sign holder after decimal	PRINT #"990.99E"-150.75; 150.75-
<	Digit holder before decimal & floats on negative number	PRINT #"<<<.00>"500; 500.00
>	Appears after decimal if negative	PRINT #"<<.00>"-50; 50.00>

The "^" formatting character translates to a decimal point upon output wherever it is located in the format field. For example, this is useful when performing monetary calculations in pennies and then translating the results to dollars and cents on output.

```
PRINT #"'999^00'"200;' 2.00'
PRINT #"'999^00'"2532;' 25.32'
PRINT #"'999^00'"12000;'120.00'
```

The comma (,) formatting character inserts a comma in the output numeric value; however, it is suppressed if there are no significant digits to the left of its position in the output value. Typically, it is used to separate groups of three decimal digits, (e.g., 1,000 and 1,000,000).

```
PRINT #"'99,990.00'"3529.87; ' 3,529.87'
PRINT #"'99,990.00'" 903; '   903.00'
PRINT #"'99,990.00'"10.2333; '    10.23'
PRINT #"'99,990.00'"100256.72; **********
```

The dollar ($) sign formatting character is used to output the dollar sign with the numeric output value. It is a digit holder and also "floats" to the position immediately to the left of the most significant digit of the output value.

```
PRINT #"'$$$.00'"25.32;'$25.32'
PRINT #"'$$$.00'".50;'  $.50'
PRINT #"'$$$.00'"100;'100.00'
PRINT #"'$$$.00'"1000;********
PRINT #"'$,$$$.00'"1.52;'   $1.52'
PRINT #"'$$,$$$.00'" 9536; '$9.536.00'
```

The "9" formatting character is used to output a signed numeric value. A minus sign (-) is output for a negative number and blank for a positive number. The "S" character is a digit holder and "floats" the sign of the numeric value to the position immediately to the left of the most significant digit of the output value.

```
PRINT #"'SSS0.00'" 208.79; ' 208.79'
PRINT #"'SSS0.00'" -20.79; ' -20.79'
```

If the user attempts to output a negative number without using the "S" formatting character, the number will be output as a positive number.

The "E" formatting character is used to output a signed numeric value with the sign appearing to the right of the decimal point. It functions only as a sign holder and is not a digit holder.

```
PRINT #"'990.00E'" 32.253; ' 32.25 '
PRINT #"'990.00E'"-32.253; ' 32.25-'
PRINT #"'990.00E'" -.50; ' 0.50-'
```

The "<" and ">" formatting characters are used in another form of outputting negative numbers. They typically are used together in the formatting string. The "<" character is a digit holder and appears before the decimal point. The ">" character appears after the decimal point and is only a sign holder. On the output of a negative number both the "<" and ">" characters are output with the string. The "<" character will float on a negative number to the position immediately to the left of the most significant digit of the output value. The ">" character will appear at its position to the right of the decimal point on a negative number. When outputting a positive number, neither the "<" nor ">" character will be output in the string.

```
PRINT #"'<<<,<<<.00>'" 1250; '  1,250.00 '
FRINT #"'<<<,<<<.00>'" -1250; ' <1,250.00>'
PRINT #"'<<<,.00>'" .20; '   20. '
PRINT #"'<<<,.00>'"-0.2; '  <.20>'
```

The following sample program further illustrates the results of print formatting. When this program is executed the user is requested to enter a numeric value and formatting string. POWER BASIC then outputs the number using the user supplied print formatting string.

```
100 DIM F(5)
110 INPUT "INPUT NUMBER"N"  FORMAT"$F(0)
120 PRINT "'"#$F(0);N"'"
130 GOTO 110

RUN

INPUT NUMBER? 1  FORMAT:  999,990.99
'      1.00'
INPUT NUMBER? 123456  FORMAT:  999,990.99
'123,456.00'
INPUT NUMBER? 529728761  FORMAT:  000-00-0000
'529-72-8761'
INPUT NUMBER? 2335.34  FORMAT:  $$$,$$$,$$$.99E
'     $2,335.34 '
INPUT NUMBER? -234.56  FORMAT:  SSSSS.99
' -234.56'
INPUT NUMBER? -2335.34  FORMAT:  $$$.$$$,$$$.99E
'     $2,335.34-'
INPUT NUMBER? 1234556  FORMAT:  999,999
'*******'
INPUT NUMBER? 123  FORMAT:  <<<,<<0.99>
'     123.00'
INPUT NUMBER? -1234  FORMAT:  <<<,<<0.99>
'<1,236.00>'
```

TAB. Output formatting can also be controlled by use of the TAB function.

Form:

```
TAB (<expression>)
```

The expression in the TAB function specifies the horizontal column position where the print item following the TAB will begin printing. The TAB function may contain any expression as its argument. The expression is evaluated and its integer portion used. If the result is greater than the line size, the specified print item will be printed on the next output line. If the column specified by the integer part of the expression has already been passed in the current print line, the TAB function will be ignored and the print item will be output at the current position in the print line. The TAB function may be used to format output into columns on the output device.

Examples:

```
10 PRINT "BIG"; TAB(20);"SPACE"
```

will generate

```
BIG                 SPACE
```

while:

```
10 PRINT TAB(20); "SPACE";TAB(1);"BIG"
```

will generate

```
                        SPACEBIG
```

In the first example, the string "BIG" is output starting in column 1. The TAB function advances the printer to column 20 and outputs the string "SPACE". In the second example, the TAB function advances the printer to column 20 and outputs the string "SPACE". The TAB (1) attempts to return the printer to column 1 in the print line. Since that column position has already been passed, the string "BIG" is output immediately following "SPACE" (the current position on the print line).

PRINT cursor control:

In conjunction with the PRINT statement, the user may position the cursor at any location on the screen. Cursor control is performed using the @ operator.

Forms:

```
[line number] PRINT @(<exp1>,<exp2>) {;} <rest of print>

[line number] PRINT @<$var> {;} <rest of print statement>
```

By using the first form, the cursor may be positioned to any coordinates specified by exp1 and exp2. Exp1 specifies the column (X) position and exp2 the row (Y) position. There are 24 lines (numbered 0 to 23) on the screen. In text mode there are 40 characters (0 to 39) on each line, in graphics mode there are 32 (0 to 31). Specifying a coordinate outside these ranges will result in the message:

```
** Invalid screen command **
```

EXAMPLES:

```
100 PRINT @(10,20);"Hello"
```

This prints the message "Hello" starting at x,y coordinates of (10,20), that is, column 10 on screen line 20.

```
200 A=10 : B=20
210 PRINT @(A,B) ; "Hello"
```

This has the same effect and shows how variables may be used for the coordinates.

The second form of cursor control uses a string containing special control letters. The string may be either a string variable or the required letters enclosed in quotes.

The valid control letters are as follows:

CODE	ACTION
B	Move cursor to beginning of line
C	Clear screen and move cursor HOME
D	Move cursor down one line
H	Move cursor to HOME (top left corner)
L	Move cursor left one character
R	Move cursor right one character
U	Move cursor up one line

Any of these codes may be preceded by a positive integer representing the number of times the code is to be repeated.

EXAMPLES:

```
10 PRINT @"C5D10R" ; "5 DOWN AND 10 TO THE RIGHT"
```

This clears the screen and moves the cursor 5 down and 10 to the right. It has the same effect as:

```
10 PRINT @"C" ; @(10,5) ; "5 DOWN AND 10 TO THE RIGHT"
```

which clears the screen and goes to coordinates 10,5.

```
10 DIM SCR(5)
20 $SCR(0)="4UB"
30 PRINT @$SCR(0); "4 UP, BEGINNING OF LINE"
```

This example shows how a string variable may be used.

Specifying X,Y coordinates will have no effect on any device other than the CORTEX main screen (device 1). The control letters will print the cursor control ASCII characters to any enabled devices.

4.2.7.3 UNIT

The UNIT statement designates the device or devices to which all subsequent PRINTed output will be sent.

Forms:

```
<line number> UNIT <expression>
              UNIT <expression>
```

The expression may be any numeric constant, variable or expression which is evaluated and its integer portion used.

If <expression> is 0 then all devices are disabled
If <expression> is -ve then the specified device is disabled
If <expression> is +ve then the specified device is enabled

The device number may be from 1 to 16. The following devices are recognized as standard by the CORTEX:

1 - CORTEX TV screen and keyboard
2 - RS232C port
3 - Cassette interface
4 - Centronics parallel printer interface

Devices 1 and 3 are part of the minimum CORTEX system. Device 2 requires the RS232 interface populated on the CORTEX board. Device 4 requires an external interface board.

When the CORTEX is turned on, and when a cold start is performed, device 1 is initialized for output. Input will be accepted from all devices except 3.

EXAMPLE:

```
10 PRINT "Testing"
20 UNIT 2
30 PRINT "Printing to RS232C"
40 UNIT -2
```

The first print at line 10 appears only on the screen. Line 20 enables the RS232C interface. The message from line 30 goes both the screen which is still enabled and to the RS232C. Line 40 turns off the RS232C again. Input is accepted from all devices.

The SYS(5) function will return information about which devices ere enabled. (See section 4.3.5).

4.2.7.4 BAUD

The BAUD statement is used to set the transmission rate of any serial I/O device in the system that uses a 9902 UART.

Forms:

```
<line number> BAUD <expression 1>,<expression 2>
              BAUD <expression 1>,<expression 2>
```

The BAUD statement will set the transmission rate of the device specified by expression 1 to the baud rate (bits per second) specified by expression 2. Devices in the standard CORTEX that will respond to this statement are:

```
2 - RS232C port
3 - Cassette interface
```

Expression 2 must evaluate to a rate between 75 and 100,000 baud. When the CORTEX is turned on or a cold start performed all 9902's in the system are initialized to 300 baud. The serial I/O uses 7 data bits, even parity and 2 stop bits.

The statement:

```
BAUD 3,<baud rate>
```

also implicitly enables input from device 3, the cassette interface. Pressing ESCAPE disables input from the cassette interface.

EXAMPLE:

```
BAUD 2,19200
```

This sets the RS232C port to 19200 baud.

4.2.7.5 MOTOR

Forms:

```
<line number> MOTOR <expression>
              MOTOR <expression>
```

The MOTOR statement is used to the tape recorder motor on and off via the remote switch connection. If <expression> is 0 the motor is turned off, if it is non-zero the motor is turned on. When using this in a program enough time must be allowed for the tape to speed up before data is output.

EXAMPLE:

```
10 UNIT 3
20 MOTOR 1
30 WAIT 100
40 PRINT "This is output to the cassette"
50 WAIT
60 WAIT 5
70 MOTOR 0 : UNIT -3
```

Line 10 enables output to the cassette and line 20 turns on the motor. Line 30 then waits for one second for the tape to speed up. Line 40 outputs to the cassette (as well as to the screen) and line 50 waits for this to complete. Line 60 waits for the last character output to reach the tape and line 70 then turns off the tape and disables output to it.

4.2.7.6 BASE

NOTE: An understanding of the CORTEX hardware is required to successfully use any of the CRU statements and functions.

The BASE statement sets the CRU base address for subsequent CRU operations.

Form:

```
<line number> BASE <expression>
              BASE <expression>
```

The BASE statement evaluates the expression and sets the CRU base address to the result for use by the CRB and CRF functions. The CRB function addresses bits within +127/-128 of the evaluated base address. The CRF function transfers bits using the evaluated base address as the starting CRU address.

The CRU provides a maximum of 4096 input and output lines that may be individually selected by a 12-bit address. The 12-bit address used by the CRU instructions is actually located in bits 3 through 14 of a workspace register. The evaluated expression of the BASE statement is loaded into the entire 16-bits of this workspace register. Therefore, the BASE expression should evaluate to twice the actual (physical) CRU base address desired since only bits 3 through 14 are used. The least significant bit of the BASE expression value is ignored for CRU operations. Therefore, all expressions should evaluate to an even number. The range of valid expressions is from 0 to 8190 (hexadecimal 1FFE).

Examples:

```
10 BASE 64
20 CRF(0)=-1
30 BASE 100
40 CRB(-1)=0
```

Statement 10 sets the CRU BASE address to 64 (physical address of 32), and statement 20 outputs a 16-bit -1 value. Statement 30 sets the CRU BASE address to 100 (physical address of 50, and statement 40 sets the CRU bit displaced -1 from the base (physical address of 49) to zero.

4.2.8 Timing Statements

4.2.8.1 TIME

The TIME statement 1s used to set, display, or store the 24 hour time-of-day clock.

Forms:

```
<line number> TIME <exp>,<exp>,<exp>
              TIME <exp>,<exp>,<exp>
<line number> TIME <string variable>
              TIME <string variable>
<line number> TIME
              TIME
```

The TIME statement is used with the expression list to set and start the time of day clock. The form of the expression is as follows:

```
TIME HH,MM,SS
```

where

M = hours, M = minutes, S = seconds

The clock is set up as a 24-hour clock with times ranging from 00:00:00 to 23:59:59. Initialization of the clock is valid at any point in the program. Its value may also be reinitialized at any point.

Examples:

```
   TIME 10,27,30 (in keyboard mode)
   TIME 3,5,0 (in keyboard mode)
10 TIME 21,8,15
```

The second form of the TIME statement enables storing the current time of day in a string variable. This is useful for recording occurrence time of significant events in a user's application program.

Example:

```
 10 DIM T(3)
 20 TIME 11,4,0
      .
      .
100 TIME $T(0)
120 PRINT $T(0)
130 STOP

RUN
   11:04:37
   STOP AT 130
```

The time of day may be directly displayed at any point within the program. It may also be displayed from the keyboard when in idle mode by using the third form of the TIME statement. The time of day will be displayed in the following format:

HH:MM:SS

Examples:

```
    TIME 9:31:23 (in keyboard mode)
 10 TIME 11,4,0
      .
      .
      .
100 TIME
110 STOP

RUN

    11:04:37
    STOP AT 110
```

4.2.8.2 WAIT

Forms:

```
[line number] WAIT <expression>         (a)
[line number] WAIT                      (b)
```

Format (a) of the WAIT statement causes the program to delay execution for (expression * 0.01) seconds.

Format (b) of the wait statement causes the program to delay execution until all current output has completed.

When the WAIT statement is executing all normal input and output operations are still active and the user may continue typing in at the keyboard without keystrokes being missed. Program execution may be aborted by pressing the 'ESCAPE' key providing that it has not been disabled via the 'NOESC' statement.

EXAMPLES:

```
100 BAUD 2,110 ! SET UP THE PRINTER FOR A TELETYPE
110 UNIT 2     ! ENABLE IT FOR OUTPUT
120 PRINT "EXECUTION BEGINS IN 1 SECOND"
130 WAIT       ! MAKE SURE THE PRINTER HAS FINISHED
140 WAIT 100   ! NOW WAIT THE 1 SECOND
150 UNIT -2    ! DISABLE THE PRINTER
160 REM *** NOW EXECUTE THE PROGRAM ***
999 END
```

The program sets up the RS232 port as a very slow printer (about 10 characters per second) and enables output to it. The message will take some time to print out, and program execution would normally continue while the printer was still going. The WAIT holds the program until the message has been output. The WAIT 100 then holds for a further one second.

4.2.9 RANDOM

The RANDOM statement randomizes the seed for the pseudo-random number generator.

Forms:

```
<line number> RANDOM <expression>
              RANDOM <expression>
```

The RANDOM statement is used in conjunction with the RND function. The RND function returns the next number in the random number sequence. It returns this value when requested and replaces it with the next random number. The RANDOM statement is used to change the random number seed and therefore the sequence of pseudo-random numbers.

The random seed is set to a constant value when POWER BASIC is first initialized so that the RND variable will always return the same sequence of numbers to facilitate program debugging. After the debugging phase, the RANDOM statement may be used to alter this sequence.

The RANDOM statement is used to set the seed to a specific or arbitrary value. The expression is evaluated and the result used as the seed of the random number generator. The expression may be any valid POWER BASIC expression. The evaluated expression must be within the limits of -32768 and 32767 or a fix error will result. The sequence of numbers generated by a specific seed value will always be the same. This is useful for debugging and testing an application program with a predetermined seed value. Arbitrary seed values may be generated by the user by using combinations of variables and functions (including the RND function) within the expression.

Examples:

```
10 RANDOM 220
20 RANDOM RND
30 RANDOM RND * MEM(X)
```

A completely random number may be generated by the sequence:

```
T=TIC(0)
RANDOM T-32767*INT(T/32767)
```

The random number seed depends on the number of clock ticks (10ms) since the computer was turned on. The additional calculation is needed to reduce the tic value to a 16 bit integer acceptable to RANDOM.

4.2.10 ESCAPE and NOESC

The ESCAPE and NOESC statements provide capability to enable or disable the escape key to interrupt program execution.

Forms:

```
<line number> ESCAPE
<line number> NOESC
```

The ESCAPE statement enables the terminal device escape (or break) key to interrupt program execution. When the escape key is struck program execution terminates upon completion of the current statement line. Keyboard sampling during the RUN mode is performed only between statement lines. Caution should be observed when certain statement constructions are used. For example, the FOR and NEXT statements should not appear in the same statement line, because a statement line is autonomous. Once the FOR/NEXT line begins execution, it cannot be interrupted by using the escape key. It can be interrupted only if the end condition of the FOR/NEXT loop is met, or if the user reinitializes the system via the reset switch on the CPU board.

The NOESC statement disables the terminal device escape (or break) key from interrupting program execution.

The ESCAPE statement is used during program development and debug. The NOESC statement is used for time critical application programs or in a production environment where it is disadvantageous for the user to interact with POWER BASIC in a non-program controlled mode.

Examples:

```
10 ESCAPE
10 NOESC
```

4.2.11 CALL

Form:

```
[line number] CALL ["name",] <address> [,<par1>,...<par12>]
```

The CALL statement is used to call external subroutines written in assembly language.

The name is optional and is the name of the routine called. It is not used by BASIC and is simply for documentation purposes.

The address is the entry point of the routine, and may be specified in decimal or hexadecimal. Up to 12 parameters may be passed, separated by commas. These are evaluated as 16 bit 2's complement integers between -32768 and 32767.

The parameters are stored in R0 to R11 of the routines workspace. R12 contains the number of parameters passed. R13 to R15 should be restored to their entry values before returning via a RTWP instruction.

If it is desired to pass variables that take more than 16 bits (such as arrays or strings) then a pointer should be passed by entering "ADR(variable)" as the parameter. This method is also used to allow an assembly language subroutine to return information to BASIC.

Space may be reserved for assembly language routines by using the NEW command. Assembly language routines may be written using the monitor, which is accessed by the MON command.

Example:

```
10 CALL "MYROUTINE" , 6200H , 42 , A , ADR($C[0])
```

This statement calls a routine from address 6200H and passes 3 parameters. The first is the number 42, the second the variable A as a 16 bit integer and the third the address of the dimensioned string variable $C(0).

4.2.12 Color Graphics Statements

4.2.12.1 TEXT

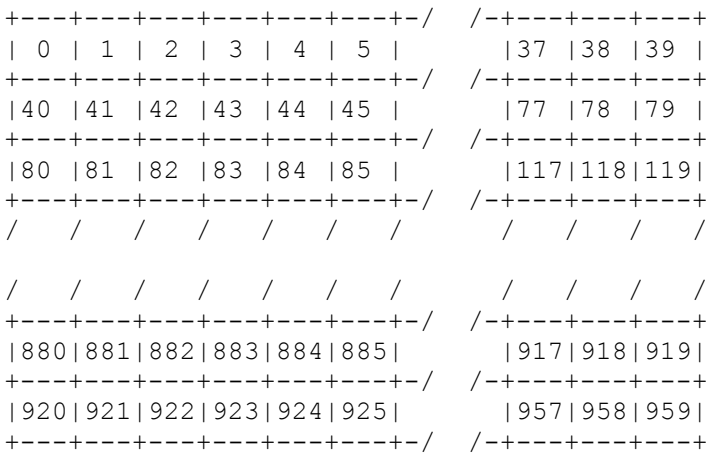
Form:

[line number] TEXT

The TEXT statement causes the video display processor to be initialized to provide 24 lines of 40 characters each. The screen will be cleared to the current background color and the text cursor positioned to 0,0 (top left hand corner). Any characters that are then printed will be in the current foreground color. This command also resets the character set to that currently contained in RAM. This command needs to be executed after the character set has been modified via the CHAR statement to cause the new character set to be used for text.

After execution of the TEXT statement the video display processor is in text mode. In text mode all characters on the screen are in the foreground color. Printing beyond the bottom of the screen causes the screen to scroll up. The character cell numbers as used by the SGET and SPUT statements are shown below.

CHARACTER CELLS - TEXT MODE



CHARACTER CELL No.= HORIZONTAL POSITION + 40*VERTICAL POSITION

In TEXT mode there are 24 rows (0 to 23) of 40 chars (0 to 39)

4.2.12.2 GRAPH

Form:

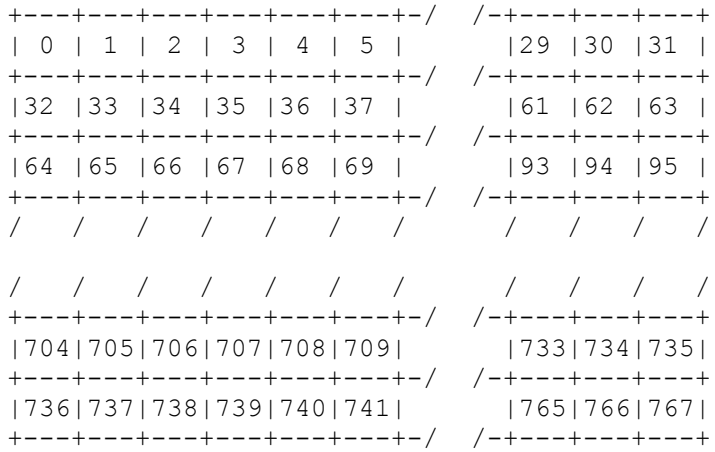
[line number] GRAPH

The GRAPH statement causes the video display processor to be initialized to provide a graphics display of 256 by 192 pixels. Normal text operations may still be performed but there are only 32 characters on a line. The screen will be cleared to the current background color and the text cursor positioned to 0,0 (top left hand corner). The text cursor is invisible in graphics mode. Any characters that are then printed will be in the current foreground color.

In graphics mode, there is an additional invisible graphics cursor.

After execution of the GRAPH statement, the video display processor is in graphics mode. Text on the screen remains in the color it was when it was printed. Printing beyond the bottom of the screen rolls round to the top. The character cell numbers as used by the SGET and SPUT statements are shown below.

CHARACTER CELLS - GRAPH MODE



CHARACTER CELL No.= HORIZONTAL POSITION + 32*VERTICAL POSITION

In GRAPH mode there are 24 rows (0 to 23) of 32 chars (0 to 31).

PIXEL CO-ORDINATES

In graphics mode there are 256 pixels horizontally (0 to 255) and 192 pixels (0 to 191) vertically. (0,0) is top left corner of the screen. These coordinates are used by the PLOT, UNPLOT and SPRITE statements.

The foreground and background colors for each group of 8 pixels horizontally is definable. (i.e.: only 2 different colors per horizontal bar of 8 pixels). This restriction does not apply to pixels colored by sprites.

Although there are only 192 pixels vertically on the screen, the vertical coordinate may be varied between 0 and 255. Pixels between 191 and 255 are "off screen". These screen locations may be used for hiding sprites without having to change their shape or color. For the SPRITE statement, the vertical coordinates are offset by 1 pixel. A vertical coordinate of -1 is at the top of the screen.

4.2.12.3 COLOUR

Form:

[line number] COLOUR [expression1] [,expression2]

The COLOUR statement is used to set the current foreground and background colors. In text mode, this immediately affects the whole screen. In graphics mode, it only affects subsequent operations.

Expression 1 should evaluate to an integer between 0 and 15 to set the foreground color. Expression 2 should evaluate to an integer between 0 and 15 to set the background color. The color codes are given below.

If the foreground color only is given then the current background color is used. If COLOUR is used with no parameters, then the default of dark blue on cyan is used.

COLOR CODES

CODE	COLOR	CODE	COLOR
0	TRANSPARENT	8	MEDIUM RED
1	BLACK	9	LIGHT RED
2	MEDIUM GREEN	10	DARK YELLOW
3	LIGHT GREEN	11	LIGHT YELLOW
4	DARK BLUE	12	DARK GREEN
5	LIGHT BLUE	13	MAGENTA
6	DARK RED	14	GREY
7	CYAN	15	WHITE

EXAMPLE:

```
COLOUR 12,11
```

This sets the foreground color to dark green and the background to light yellow.

ADDENDUM: there is also an undocumented Basic command - SWAP - which does a screen color substitution in GRAPH mode. To use it, first there is a bug in one of the jumps in the code that needs to be corrected: MWD(3390H)=1308H. A color substitution table then has to be set up, this is stored in the 16 bytes from >00A4 to >00B3. Initially these contain the values >00 to >0F. The SWAP command goes through the screen color table and if a pixel currently has the color code 0 then it replaces that with the color specified by the color code in the first byte of the table. If the current color code is 1 then it replaces that with the color specified by the color code in the second byte of the table, and so on. The color codes are as specified above.

4.2.12.4 PLOT

Forms:

```
      [line number] PLOT <exp1>,<exp2>                    (a)
      [line number] PLOT TO <exp3>,<exp4>                 (b)
      [line number] PLOT <exp5>,<exp6> TO <exp7>,<exp8>   (c)
```

The PLOT statement is used to plot points and draw lines. PLOT is available only in graphic mode. If it is attempted whilst in text mode the video display processor will be switched to graphic mode before plotting.

The simplest form of the PLOT statement (a) will plot a point in the current foreground color. The point plotted has a horizontal coordinate of <exp1> and a vertical coordinate of <exp2>. After the point has been plotted the graphic cursor address will be set to this point.

The next form of the PLOT statement (b) will cause a line to be drawn in the current foreground color from the current graphic cursor location to the point having a horizontal coordinate of <exp3> and a vertical coordinate of <exp4>. After the line has been drawn the graphic cursor will be set to <exp3>,<exp4>.

The next form of the PLOT statement (c) will cause a line to be drawn in the current foreground color from the point having a horizontal coordinate of <exp5> and a vertical coordinate of <exp6> to the point having a horizontal coordinate of <exp7> and a vertical coordinate of <exp8>. After the line has been drawn the graphic cursor will be set to <exp7>,<exp8>.

Both forms (b) and (c) may be followed by a 'TO <exp>,<exp>' section which will continue plotting a line to the point <exp>,<exp>. This extension may be repeated as long as there is room on the BASIC line to do so.

The coordinates used for PLOT are pixel coordinates, explained under the GRAPH statement.

NOTE :- When PLOTting lines the background color should remain the same as when the screen was last cleared (e.g., by a GRAPH statement). If the background color is not the same, the limitation of horizontal color resolution causes blocks to be set to the current background color.

EXAMPLES:

```
100 COLOUR 4,7           ! DARK BLUE ON A LIGHT BLUE BACKGROUND
110 GRAPH                ! GO INTO GRAPHIC MODE AND CLEAR SCREEN
120 PLOT 10,20           ! SET THE POINT HOR=10,VERT=20
130 PLOT 40,50 TO 100,105    ! DRAW A LINE FROM 40,50 TO 100,105
140 COLOUR 15            ! NOW DRAW WHITE LINES
150 PLOT TO 200,15       ! DRAW A LINE FROM 100,105 TO 200,15
155 REM
160 REM NOW DRAW A WHITE BOX ROUND THE WHOLE THING
165 REM
170 PLOT 0,0 TO 255,0 TO 255,191 TO 0,191 TO 0,0
175 WAIT 100
180 COLOUR 1             ! CHANGE TO BLACK LINES
185 REM
190 REM NOW DRAW VERTICAL BARS
195 REM
200 FOR I=20 TO 250 STEP 8
210 PLOT I,1 TO I,190
220 NEXT I
230 END
```

4.2.12.5 UNPLOT

Forms:

```
      [line number] UNPLOT <exp1>,<exp2>                    (a)
      [line number] UNPLOT TO <exp3>,<exp4>                 (b)
      [line number] UNPLOT <exp5>,<exp6> TO <exp7>,<exp8>   (c)
```

The UNPLOT statement is the opposite of PLOT. It is used to erase points and lines. UNPLOT is available only in graphic mode. If it is attempted whilst in text mode the VDP will be switched to graphic mode before unplotting.

The simplest form of the UNPLOT statement (a) will unplot a point having a horizontal coordinate of <exp1> and a vertical coordinate of <exp2>. After the point has been unplotted the graphic cursor address will be set to this point.

The next form of the UNPLOT statement (b) will cause a line to be undrawn from the current graphic cursor location to the point having a horizontal coordinate of <exp3> and a vertical coordinate of <exp4>. After the line has been undrawn the graphic cursor will be set to <exp3>,<exp4>.

The next form of the UNPLOT statement (c) will cause a line to be undrawn from the point having a horizontal coordinate of <exp5> and a vertical coordinate of <exp6> to the point having a horizontal coordinate of <exp7> and a vertical coordinate of <exp8>. After the line has been drawn the graphic cursor will be set to <exp7>,<exp8>.

Both forms (b) and (c) may be followed by a 'TO <exp>,<exp>' section which will continue unplotting a line to the point <exp>,<exp>. This extension may be repeated as long as there is room on the BASIC line to do so.

The coordinates used for UNPLOT are pixel coordinates. These are explained under the GRAPH statement.

NOTE :- When UNPLOTting lines the current foreground and background colors are not important, all the points that are unplotted will be set to the background color already associated with that point.

EXAMPLES:

```
100 COLOUR 4,7     ! DARK BLUE ON A LIGHT BLUE BACKGROUND
110 GRAPH          ! GO INTO GRAPHIC MODE AND CLEAR SCREEN
120 REM              NOW DRAW A BOX
130 PLOT 0,0 TO 255,0 TO 255,191 TO 0,191 TO 0,0
140 WAIT 300       ! WAIT THREE SECONDS AND THEN REMOVE IT
150 UNPLOT 0,0 TO 255,0 TO 255,191 TO 0,191 TO 0,0
160 END
```

4.2.12.6 SHAPE

Form:

```
[line number] SHAPE <arg1>,<arg2>,<arg3>,<arg4>,<arg5>
```

The SPUT and SPRITE commands use predefined shapes. The SHAPE statement is used to define these shapes.

A shape table is used to store these shapes and has 256 entries (0 to 255). An entry is in the form of 4 integers, each of 16 bits. Each bit represents one pixel on the screen, arranged in an 8 by 8 matrix as shown:

```
 First 16 bits/  * * * * * * * *
              \  * * * * * * * *
Second 16 bits/  * * * * * * * *
              \  * * * * * * * *
 Third 16 bits/  * * * * * * * *
              \  * * * * * * * *
Fourth 16 bits/  * * * * * * * *
              \  * * * * * * * *
```

Each 16 bits is arranged with the most significant bit at top left and least significant at bottom right of the group of 16.

The 5 arguments of SHAPE are:

```
arg1 : Shape table entry to use (0 to 255)
arg2 : integer (16 bits) pattern of 1st & 2nd row of shape
arg3 : integer (16 bits) pattern of 3rd & 4th row of shape
arg4 : integer (16 bits) pattern of 5th & 6th row of shape
arg5 : integer (16 bits) pattern of 7th & 8th row of shape
```

It is often convenient to express the arguments as hexadecimal constants. This means that each character in the argument maps to four pixels in the shape.

EXAMPLE:

```
10 SHAPE 42,0FFFFH,0FFFFH,0,0
```

This defines shape 42 as a block which is 8 pixels by 4.

4.2.12.7 SPUT

Form:

```
[line number] SPUT <character cell number>,<what to put there>
```

The SPUT (Screen PUT) statement is used to transfer characters or shapes to the screen. The effect depends on the current screen mode.

In text mode "<what to put there>" is the ASCII code of the character to print. This may be obtained by using the ASC function.

In graphics mode "<what to put there>" is a shape table entry (0 to 255) for the shape to display on the screen. The SHAPE statement is used to define these shapes.

The character cell number is 0 to 959 in text mode and 0 to 767 in graphics mode. See the TEXT and GRAPH statements for how these map onto the screen.

Example:

```
10 TEXT                 ! Text mode
20 CH=ASC("X")          ! Get the ASCII code for an X
30 SPUT 80,CH           ! Put X in cell number 80
35 WAIT 500
40 GRAPH                ! Graphics mode
50 SHAPE 1,0,5A5AH,0,0 ! Define shape 1 as a row of dots
60 SPUT 140,1           ! Display dots in cell 140
```

4.2.12.8 SGET

Form:

[line number] SGET <screen cell number>,<variable>

The SGET (Screen GET) statement is used for reading information from the video display. As for SPUT, the effect depends on the display mode.

The variable must be a numeric variable. The cell number is from 0 to 959 in text mode and 0 to 767 in graphics mode. The mapping of these cells to the display is explained under the TEXT and GRAPH statements.

In text mode the variable will receive the ASCII code of the character in the specified display cell.

In graphics mode, the variable specifies the shape table entry to receive the current shape in the specified display cell. In graphics mode only, a constant may be used as the second argument.

EXAMPLES:

A)
```
10 TEXT
20 CH=ASC("X")   ! Get the ASCII code for an X
30 SPUT 160,CH   ! Put X at start of line 4
40 C=0           ! Declare variable C
50 SGET 160,C    ! Get code of char at start of line 4
60 PRINT C       ! Print ASCII code
```

B)
```
50 GRAPH
60 SHAPE 1,0,5A5AH,0,0 ! Define shape 1 as a row of dots
70 SPUT 128,1    ! Display dots at start of line 4
80 SGET 128,10   ! Put shape at start of line 4 in shape 10
90 SPUT 144,10   ! Display same shape half way along line 4
```

4.2.12.9 CHAR

Forms:

```
[line number] CHAR <ASCII code>,<exp1>,<exp2>,<exp3>
[line number] CHAR
```

The CHAR statement is used to redefine the character set used for printing text.

The first argument is the ASCII code of the character to be redefined. Exp1 to exp3 evaluate to 16 bit integers. This forms a 48 bit pattern that maps onto the 6*8 character cell:

```
  exp1 -> * * * * * *
  exp1 -> * * * * * *     The 48 bits of the pattern
exp1/2 -> * * * * * *     start from top left with
  exp2 -> * * * * * *     the MSB of exp1 and work
  exp2 -> * * * * * *     down to bottom right with
exp2/3 -> * * * * * *     the LSB of exp3.
  exp3 -> * * * * * *
  exp3 -> * * * * * *
```

In text mode the change will take effect next time a TEXT statement is executed. In graphic mode the change takes effect immediately.

A CHAR statement with no arguments will load the standard character set back again.

EXAMPLE:

```
10 C = ASC("d")      ! Find ASCII code of char to change
20 CHAR C , 0820H,8628H,0A380H
30 TEXT              ! Effect the change
40 PRINT "ddddddddd" ! Look at the result
```

This example changes the small "d" to look like lower case instead of a small capital.

4.2.12.10 SPRITE

Form:

```
[line number] SPRITE <arg1>,<arg2>,<arg3>[,<arg4>,<arg5>]
```

Where:

arg1 = sprite number (0 to 31)
arg2 = horizontal pixel coordinate, top left of sprite
arg3 = vertical pixel coordinate, top left of sprite
arg4 = shape number to use for the pattern (0 to 255)
arg5 = sprite color (0 to 16)

A sprite is a shape which may be moved to any point on the screen by specifying the pixel coordinates. Up to 32 sprites maybe on the screen at any one time.

Each of the 32 sprites has it's own plane, numbered from 0 to 31. When two sprites overlap, the lower numbered sprite passes over the top of the higher numbered. All sprites pass over background graphics end text. This feature may be used to build up a 3D effect. Sprite 0 appears closest to the viewer with sprite 31 farthest away.

The SPRITE statement is used to display a sprite on the screen. The first argument specifies which of the 32 sprites is to be used and hence its priority over other sprites.

The second and third arguments specify where to put the sprites in pixel coordinates. These are explained under the GRAPH statement.

The shape of the sprite is defined by the SHAPE statement. Which shape to use is defined by argument 4. The color of the sprite, from 0 to 15, is defined by the last argument. See the COLOUR statement for a list of colors.

If the last two arguments are omitted, the sprite in the specified plane is moved without changing shape or color.

Normally each bit of the shape is mapped onto one pixel of the screen (see SHAPE statement). This may be altered by using the MAG statement. Multicolored shapes may be built up by overlaying sprites of different colors.

NOTE: Sprites must be first used in order from 0 upwards. A sprite will have no effect unless the previous sprite plane sprite has been used. If a higher sprite plane is needed first, the lower ones must first be initialized to an off screen position or empty shape. Only 4 sprites may be active together on one screen line.

EXAMPLE:

```
 10 COLOUR 1,0
 20 GRAPH
 30 COLOUR
 40 A=01038H
 50 B=0547CH
 60 C=09244H
 70 D=04482H
 80 NME=42
 90 SHAPE NME,A,B,C,D
100 MAG 1,0
110 FOR X=0 TO 1000 STEP 16
120 FOR Y=0 TO 175
130 SPRITE 0,X,Y,NME,6
140 SPRITE 1,X+16,Y,NME,10
150 SPRITE 2,X+32,Y,NME,4
160 SPRITE 3,X+48,Y,NME,12
170 NEXT Y
180 NEXT X
```

This program first defines a shape. Then, using the MAG statement to double their size, moves 4 different colored sprites of the defined shape across the screen.

Sprites at the screen edges

As sprites are moved to the edges of the screen they may be 'trickled' onto and off of the display. This means that the sprite appears or disappears gradually.

Using horizontal coordinates for the top left of the sprite approaching 255 will trickle the sprite on and off the right hand side of the screen. Vertical coordinates approaching 191 will trickle on and off the bottom of the screen. To trickle at the top, use coordinates working back from -1.

Since all 255 possible horizontal coordinates are used for on screen positions, it is not possible to trickle at the left by specifying coordinates. This is achieved by a special mode. If the color of the sprite is set to the required color plus 128 the whole sprite is shifted 32 pixels to the left. In this mode, the sprite may trickle at the left by using horizontal coordinates between 0 and 31.

4.2.12.11 MAG

Form:

[line number] MAG <magnification>,<definition size>

The MAG statement is used to define how sprites will be displayed on the screen.

The magnification argument allows the sprite dimensions to be doubled (area quadrupled) with a corresponding reduction in resolution. The definition size allows a similar increase in size using more shape table data to retain resolution. The combination of the two allows sprites to be magnified up to four times (16 times area).

If the sprite magnification is zero every bit in the shape definition for the sprite will be displayed as one pixel, if the sprite magnification is not zero then each bit in the shape definition will be displayed as two pixels horizontally and two pixels vertically.

If the sprite definition size is zero then one shape table entry will be used to build the sprite, if it is non-zero then four shape table entries will be used to build the sprite. The shape table entries used for the sprite in this mode must start on a four entry boundary.

For the large sprite definition the four shape table entries used are joined in the following way to build a 16x16 point sprite.

```
+---------+---------+
| shape n |shape n+2|   where "n" is the shape number
+---------+---------+   given to the sprite
|shape n+1|shape n+3|   statement. Valid values
+---------+---------+   of "n" are 0,4,8,12,16 etc.
```

See the SPRITE command for an example of the use of MAG.

4.2.13 ENTER

Form:

```
[line number] ENTER <string>
```

The ENTER statement is used to enter new program lines from the program. The string may be a string constant in quotes or a string variable.

As when entering programs at the keyboard, a statement without a line number is executed immediately. Syntax checking is performed as normal. If the line number already exists, it is replaced. Entering just a line number deletes that line.

EXAMPLES:

A) Simple use of the ENTER statement:

```
ENTER "500 PRINT 'HELLO'"

**NO SUCH LINE NUMBER**

LIST

500 PRINT "HELLO"
```

B) A program using ENTER to input a function:

```
 10 DIM LIN(10),IP(10)
 20 INPUT "VALUE FOR X" X
 30 INPUT "FUNCTION (Y=FN(X))" $IP(0)
 40 $LIN="100 "+$IP(0)
 50 ENTER $LIN(0)
 60 GOSUB 100
 70 PRINT Y
 80 STOP
100 REM FUNCTION REPLACES THIS LINE
120 RETURN

RUN

VALUE FOR X? 2
FUNCTION (Y=FN(X)): Y=SIN(X)*COS(X)
-0.3784
Stop at 80
```

This program allows the user to type in a function (in valid BASIC syntax) which is then evaluated. Care should be taken with programs that modify themselves in this way. They are extremely difficult to debug.

4.2.14 TON and TOF

Forms:

```
[line number] TON
[line number] TOF
```

Statement tracing allows the programmer to follow the execution of a BASIC program. The TON and TOF statements turn statement tracing on and off respectively. When tracing is enabled, a message is output as each line is executed. This is of the form:

```
Statement No. xxx
```

Where xxx is the line number. If there are further statements on the same line, the trace output will continue with:

```
Statement No. xxx.1
Statement No. xxx.2
     etc.
```

TON and TOF entered for immediate execution will allow tracing of a whole program to be enabled or disabled. Entering these statements as lines in the programs allows tracing to be turned on for just a specific part of the program.

EXAMPLE:

The example program from the ENTER statement, (previous section) would produce:

```
TON
RUN

Statement No. 10
Statement No. 20
VALUE FOR X? 2
Statement No. 30
FUNCTION (Y=FN(X)): Y=SIN(X)*COS(X)
Statement No. 40
Statement No. 50
Statement No. 60
Statement No. 100
Statement No. 120
Statement No. 70
-0.3784
Statement No. 80
```

NOTE: Statement tracing has some limitations. Care should be used when interpreting the results of jumps into and out of the middle of multiple statement lines.

4.3 BASIC FUNCTIONS

4.3.1 Mathematical Functions

4.3.1.1 ABS

Form:

```
[line number] <variable> = ABS(<expression>)
```

The absolute value function (ABS) obtains the absolute value of a positive or negative number. The argument entered following the function name is the variable name or numeric value for which the absolute value is required. The function returns a non-negative argument unaltered and returns the absolute value of a negative argument.

Example:

```
10 INPUT X
20 PRINT SQR(ABS(X))
30 STOP
```

4.3.1.2 ATN

Form:

```
[line number] <variable> = ATN(<expression>)
```

The argument entered following the function name is the ratio representing a tangent function. The function returns the corresponding angle in radians. Multiply the number of radians by 180/3.14159265 (Pi) to obtain the angle in degrees.

```
10 INPUT X
20 D = ATN(X)*(180/3.14159265)
30 PRINT D
40 STOP
```

Executing the above example produces:

```
? 5.9246
80.419473251
```

4.3.1.3 SIN and COS

Forms:

```
[line number] <variable> = SIN(<expression>)
[line number] <variable> = COS(<expression>)
```

The argument entered following the function name represents an angle in radians. When the angle is measured in degrees, multiply the number of degrees by 3.14159265 (Pi) /180 to obtain the angle in radians. The function determines the quadrant corresponding to the argument and returns the function value.

Example:

```
10 INPUT N
20 PRINT SIN(N);COS(N);
30 STOP
```

Executing the above example produces:

```
? 1.25
0.94898461936      0.31532236237
```

4.3.1.4 EXP

Form:

```
[line number] <variable> = EXP(<expression>)
```

The argument entered following the function name is an exponent of "e" (the base of natural logarithms). The function returns the value of "e" raised to the power specified in the argument.

Example:

```
10 INPUT E
20 PRINT EXP(E)
30 STOP
```

Executing the previous example produces:

```
? 35
1.5860134525E15
```

4.3.1.5 FRA

Form:

```
[line number] <variable> = FRA(<expression>)
```

The fractional part function returns the fractional portion of the expression. The expression entered after the function name is the value for which the fractional part is required.

EXAMPLE:

```
10 A= 5.23479
20 B= FRA(A)
30 PRINT A,B

RUN

5.23479        0.23479
```

4.3.1.6 INT

Form:

```
[line number] <variable> = INT(<expression>)
```

The integer part function returns the integer portion of the expression. The INT function is useful in modular arithmetic and for correcting errors resulting from truncation or rounding of functions. The expression entered following the function name is the value for which the integer portion is required.

EXAMPLE:

```
10 INPUT Y
20 IF INT(Y/2) <> Y/2 THEN LOTO 50
30 PRINT "Y IS AN EVEN NUMBER"
40 STOP
50 PRINT "Y IS AN ODD NUMBER"
60 STOP

RUN

? 75
Y IS AN ODD NUMBER
Stop at 60
```

NOTE: There is no rounding carried out on the argument before the INT operation is carried out. For example:

```
INT(1.9999999999) = 1 NOT 2
```

Whilst this is logically correct, it may cause problems with expressions such as

```
INT(X^Y)
```

due to the rounding errors in exponentiation and other mathematical functions.

4.3.1.7 LOG

Form:

```
[line number] <variable> = LOG(<expression>)
```

The argument entered following the function name is the value for which the natural logarithm (base e) is required. The function returns the natural logarithm of the argument. Attempts to find the logarithm of a non-positive argument will result in an error.

Example:

```
10 INPUT L
20 FRINT LOG(L)
30 STOP
```

Executing the above example produces:

```
? 5280
8.5716813767
```

4.3.1.8 MOD

Form:

```
[line number] <variable> = MOD(<expression1>,<expression2>)
```

The MOD function performs modulus division between the two arguments. The result is the remainder of expression1 divided by expression2. The arguments are truncated to integers, the result is therefore always an integer.

EXAMPLE:

```
10 PRINT M0D(9,7)

RUN

2
```

4.3.1.9 SGN

Form:

```
[line number] <variable> = SGN(<expression>)
```

The sign function returns the sign of the expression. The result is as follows:

```
If <expression> is negative the result is -1
If <expression> is zero the result is 0
If <expression> is positive the result is 1
```

EXAMPLE:

```
10 V=42
20 A=SGN(V) : B=SGN(-567.45) : C=SGN(0)
30 PRINT A;B;C

RUN

1 -1 0
```

4.3.1.10 SQR

Form:

```
[line number] <variable> = SQR(<expression>)
```

The square root (SQR) function returns the square root value of the specified argument. The argument entered following the function returns the square root of the argument. An error message is produced if the argument is negative.

Example:

```
10 INPUT K
20 PRINT SQR(K)
30 STOP
```

Executing the above example produces:

```
? 2
1.4142135623
```

4.3.2 String Functions

Throughout this section, <string> may be a literal string in quotes or a string variable starting with "$". Section 6 gives More details of string operations.

4.3.2.1 ASC

Form:

```
[line number] <variable> = ASC(<string>)
```

The ASCII character conversion (ASC) function returns the decimal ASCII numeric value of the first character of the specified string function.

Example:

```
10 $A="B"
20 B=ASC [$A]
30 $C=%B+020H
40 D=ASC [$C]
50 PRINT $A,B,$C,D
60 STOP

RUN

B              66              b              98
STOP AT 60
```

4.3.2.2 LEN

Form:

```
[line number] <variable> = LEN(<string>)
```

The length (LEN) function returns the number of non-null characters starting at the evaluated address. The argument of the LEN function must be specified as a string by either the "$" or "string constant" operators.

Example:

```
10 $I="ABC"
20 J=LEN($I)
30 K=LEN("ABCDEFGHIJKLMNOP")
40 PRINT J,K
50 STOP
```

Executing the above example produces:

```
3     16
```

4.3.2.3 MCH

Form:

```
[line number) <variable> = MCH(<string1>,<string2>)
```

The character match function (MCH) returns the number of characters to which the two strings agree. A value of zero indicates no match.

Example:

```
10 $C="ABCD"
20 M=MCH("AB",$C)
30 PRINT M
40 STOP
```

Executing the above example produces:

```
2
```

4.3.2.4 POS

Form:

```
[line number] <variable> = POS(<string1>,<string2>)
```

The position function returns the character position of string1 in string2. A character position of 0 indicates that string2 does not contain string1.

EXAMPLE:

```
10 $C="ABCD"
20 S=POS("BC",$C)
30 PRINT S
40 STOP

RUN

2
```

4.3.3 Input and Output Functions

4.3.3.1 CRB

NOTE: An understanding of the CORTEX hardware is required to successfully use any of the CRU functions. See also BASE statement.

Forms:

```
[line number) <variable> = CRB(<expression>)
(line number] CRB(<expression1>) = <expression2>
```

A CRU bit, addressed relative to a base displacement, is either read or stored according to program context. The displacement ranges from -128 to +127. The function returns a 1 if the CRU bit is set, and a 0 if not set. Likewise, the selected CRU bit is set to 1 if the assigned value is non zero and to 0 if the assigned value is zero. For example:

```
CRB(10)=0
```

will clear the tenth bit relative to the base, while

```
CRB(11)=1 or CRB(11)=345
```

will set the eleventh bit on. Also,

```
IF CRB(5) THEN J=4
```

4.3.3.2 CRF

NOTE: An understanding of the CORTEX hardware is required to successfully use any of the CRU functions. See also BASE statement.

Forms:

```
[line number] <variable> = CRF(<expression>)
[line number] CRF(<expression1>) = <expression2>
```

The CRU field function is used to transfer up to 16 bits of data to or from the CRU. The expression specifies the number of bits to be transferred. These bits are transferred to or read from the CRU starting at the address set by the BASE statement. The specified number of bits ranges from 0 to 15. If 0, then 16 bits will be transferred. For example:

```
CRF(0) = -1
```

transfers 16 bits (hex "FFFF") to the CRU address specified by the BASE statement. While,

```
VAL = CRF(8)
```

reads 8 bits from the CRU base address and stores the result in VAL.

4.3.3.3 KEY

Form:

```
[line number] <variable> = KEY(<expression>)
```

The KEY function reads the keyboard while a program is running. It allows input to be accepted without causing the program to stop and wait for it. This is particularly useful in games applications.

When the expression is 0 the ASCII value of the last key struck is returned and the key register reset (This means another key code may now be accepted).

If the expression is non-zero, it is compared with the last key struck. If they are the same, a value of 1 is returned and the key register reset. Otherwise, a value of 0 is returned. For example:

```
I = KEY(0)
```

returns the last key struck, or a 0 if no key was pressed.

```
IF KEY(041H) THEN PRINT "A"
```

prints A if the last key entered was "A" (The ASCII code for A is 41 in hexadecimal, it could also be expressed as 65 in decimal).

EXAMPLE:

```
10 A=KEY(0)
20 IF A=0 THEN GOTO 10
30 PRINT A
40 GOTO 10
```

This will print out the values of any keys pressed.

4.3.4 Memory Functions

When manipulating memory remember that the CORTEX is a 16 bit machine:

```
1 Byte - 8 bits
1 Word - 2 bytes - 16 bits
```

4.3.4.1 ADR

Form:

```
[line number] <variable> = ADR(<variable>)
```

The ADR function returns the address in memory of the specified variable, string variable, array element or byte offset in an array element. This is used in conjunction with the LET (assignment), PRINT and CALL statements.

Valid uses are:-

```
B = ADR(<var>)
PRINT #, ADR(<var>)
CALL . . . . ., ADR(<var>)
```

where <var> can be one of the following:

```
A, $A, A(index), $A(index), or $A(index ; offset)
```

This function returns the address of the first byte (i.e., lowest memory address) for the specified variable.

EXAMPLE:

```
10 A=0 : B=0
20 CALL "GETAB" , GAB , ADR(A) , ADR(B)
30 PRINT A,B
```

This program passes the addresses of A and B to the Assembly language routine. GAB is variable previously initialized to the start address of the GETAB routine. This routine can now access A and B to return results that BASIC can print.

4.3.4.2 BIT

Forms:

```
      [line number] <variable> = BIT(<variable>,<bit>)
      [line number] BIT(<variable>,<bit>) = <expression>
```

The bit modification (BIT) function reads or modifies any bit within a variable. The function returns a 1 if the bit is set and a 0 if not set. Likewise, the selected bit is set to one if the assigned value is non-zero, and to zero if the assigned value is zero. For example:

```
      IF BIT (A, 31) THEN PRINT "ON"
```

prints "ON" if bit 3l of variable A is on; while

```
      BIT (A, 30)=1 or BIT (A, 30)=750
```

turns "on" the 30'th bit of variable A.

4.3.4.3 MEM

Forms:

```
[line number] <variable> = MEM(<address>)
[line number] MEM(<address>) = <expression>
```

The memory modification (MEM) function reads or modifies a memory location (byte) as specified by the argument. For example:

```
M = MEM(0AA00H)
```

reads the byte from location hex "AA00", while

```
MEM(0AA00H) = 15
```

stores a decimal 15 (hex "F") at location hex "AA00"

4.3.4.4 MWD

Forms:

```
[line number] <variable> = MWD(<address>)
[line number] MWD(<address>) = <expression>
```

The memory word modification function reads or modifies a memory word as specified by the argument. For example:

```
M=MWD(0FF00H)
```

reads the word from location hew "FF00", while

```
MWD(0AA00H) = 256
```

stores a decimal 256 (hex "100") in the memory word at location hex "AA00".

4.3.5 System Functions

4.3.5.1 SYS

Form:

[line number] <variable> SYS(<expression>)

The system function returns the value of the system variable defined by the value of <expression>. System variables returned are:

HELP FLAG	SYS(0)
LAST ERROR NUMBER	SYS(1)
LAST ERROR LINE No.	SYS(2)
CRU BASE ADDRESS	SYS(3)
ERROR FLAG	SYS(4)
UNIT FLAG	SYS(5)
ESCAPE DISABLE FLAG	SYS(6)
START OF USER RAM	SYS(7)
LINE EDIT ERROR FLAG	SYS(8)
VDP MODE	SYS(9)
FDC INTERRUPT FLAG	SYS(10)
VDP STATUS FLAG	SYS(11)
BASIC ENTRY POINT	SYS(12)
INTERRUPT 1 VECTOR	SYS(13)
INTERRUPT 4 VECTOR	SYS(14)
MID O VECTOR	SYS(15)
A/O VECTOR	SYS(16)
ILLEGAL OPCODE VECTOR	SYS(17)
PRINT PREPROCESSOR	SYS(18)
BREAKPOINT O VECTOR	SYS(19)

System variables 13 to 19 return information about system vectors. These are intended for the advanced user and may be used to set up service routines of various types.

4.3.5.2 SYS(0)

System function 0 returns the current line number for help on input. As explained under the INPUT statement, a line number may be specified for a help routine if invalid input is attempted.

EXAMPLE:

```
10 INPUT ?120   A
20 PRINT SYS(0)
30 STOP
120 PRINT "WRONG, TRY AGAIN" : GOTO 10

RUN

? <any number>
120
```

4.3.5.3 SYS(1)

System function 1 returns the number of the last error that occurred. It can be used in conjunction with the ERROR statement to write an error handler routine.

EXAMPLE:

```
10 ERROR 100
20 A=5/0
30 STOP
100 PRINT SYS(1)
130 STOP

RUN

28
```

The error number for divide by zero is 28 (see Appendix C).

4.3.5.4 SYS(2)

System function 2 returns the line number where the last error occurred. Taking the example program for SYS(1) and adding the line:

```
110 PRINT SYS(2)
```

we get:

```
RUN

28
20
```

i.e.,: error 28 occurred at line 20.

4.3.5.5 SYS(3)

System function 3 returns the current CRU base address as set by the BASE statement.

EXAMPLE:

```
10 BASE 0100H
20 PRINT # SYS(3)

RUN

0100H
```

4.3.5.6 SYS(4)

System function 4 returns the current line number for error processing, as set by the ERROR statement. Adding a further line to the SYS(1) example program:

```
15 PRINT SYS(4)

RUN

100
28
20
```

Error 28 occurred at line 20 and error trapping caused a jump to line 100.

4.3.5.7 SYS(5)

System function 5 returns information about which devices have been enabled or disabled with the UNIT statement. The value returned is a 16 bit integer. Each bit a corresponds to a device:

```
          MSB                                                   LSB
     BIT:   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  DEVICE:  16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```

If the device is enabled, the bit is set to 1. If it is disabled, the bit is set to 0. Standard CORTEX devices are:

1 - Keyboard and visual display
2 - RS232C port
3 - Cassette interface
4 - Centronics parallel interface

EXAMPLE:

```
10 U= SYS(5)
20 WRD=16
10 IF BIT(U,WRD+14) THEN PRINT "RS232C IS TURNED ON"
40 ELSE PRINT "RS232C IS TURNED OFF"
```

The word offset is needed because the integer is stored in the second word of U.

4.3.5.8 SYS(6)

System function 6 returns the escape disable flag. This is -1 (0FFFFH) if escape is disabled (NOESC statement has been executed) and 0 if escape is enabled (normal state or after ESCAPE statement).

EXAMPLE:

```
10 NOESC
20 PRINT SYS(8)
30 ESCAPE
40 PRINT SYS(8)

RUN

-1
0
```

4.3.5.9 SYS(7)

System function 7 returns the current lower limit of the user RAM. This is an area completely unused by CORTEX software. It is available to the assembly language programmer. The top of this area is set by the NEW command. The default value is 1000 above the start.

EXAMPLE:

```
PRINT # SYS(7)

60EEH
```

4.3.5.10 SYS(8)

System function 8 returns the state of the line editing flag. Normally lines entered with an error are presented for editing to correct the error. In this state SYS(8) will have a value of 0. If line editing has been disabled using the ERROR command, SYS(8) will have a value of -1 or FFFFH.

4.3.5.11 SYS(9)

System function 9 returns the current display mode. A value of 0 is returned for TEXT mode and -1 (0FFFFH) for GRAPH mode.

EXAMPLE:

```
10 TEXT : PRINT SYS(9)
15 WAIT 200
20 GRAPH: PRINT SYS(9)

RUN

0
-1
```

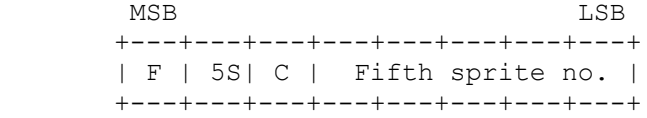
4.3.5.12 SYS(10)

System function 10 returns the state of the floppy disc interrupt flag. This indicates whether or not the floppy disc controller has requested an interrupt. It is set by the default interrupt service routine and allows operation of the floppy disc interface from a BASIC program.

If no interrupt has been received it will have a value of 0, when an interrupt is received it is set to -1. It is up to the user to reset the flag when the interrupt has been serviced.

4.3.5.13 SYS(11)

System function 11 returns the value of the video display processor internal status register. This is an 8 bit value as shown below.



F is the interrupt flag. VDP interrupts are not enabled in the CORTEX but the F flag will still be set to 1 at the end of each frame scan. It is reset to 0 when the status register is read (i.e., when SYS(11) is executed).

5S is a flag that is set when more than 4 sprites are active together on a line. The fifth sprite will not be visible if this happens. The number of the sprite that has disappeared is given by the five least significant bits of the status register. This flag and the sprite number are only valid when F is 0.

C is the coincidence flag. This is set whenever two or more sprites on the screen have overlapping pixels. The C flag is reset to 0 when the status register is read.

Note that the fifth sprite information is not valid when F is 1. If F is set, which will normally be the case, a second call to SYS(11) must be used to get the 5S information.

4.3.5.14 SYS(12)

System function 12 returns the BASIC entry point. A jump to this location from a machine code routine or the Monitor will warm start into BASIC. The correct workspace pointer is set for BASIC and the current program is not deleted.

A common use for this jump is to get back into BASIC whilst leaving the Monitor breakpoints set. The G command will clear all breakpoints.

4.3.5.15 SYS(13)

System function 13 returns the location of the interrupt level 1 vector. The vector consists of 2 words for the interrupt service routine (ISR). The first word is the ISR workspace pointer and the second is the ISR entry point.

This vector is normally not set. To add your own level 1 ISR, this vector should be set to point to your routine. Return is via a RTWP instruction.

The only hardware connected to interrupt level 1 in the standard CORTEX is the E-BUS timeout. This will only cause an interrupt if specifically enabled by the user.

4.3.5.16 SYS(14)

System function 14 returns the location of the interrupt level 4 vector. The vector consists of 2 words for the interrupt service routine (ISR). The first word is the ISR workspace pointer and the second is the ISR entry point.

This vector is normally not set. To add your level 4 ISR, this vector should be set to point to your routine. It will be called after CORTEX has checked the standard devices for interrupt. Checking is done in the following order:

1) Keyboard interrupt
2) Floppy disc interrupt
3) 9902 interrupt
4) User trap via level 4 vector

Return from the user routine is via a RTWP instruction.

4.3.5.17 SYS(15)

System function 15 returns the location of the MID 0 vector. This enables the effect of opcode 0 to be defined. The vector is one word which is the entry point for the routine. Return is via a RTWP instruction.

4.3.5.18 SYS(16)

System function 16 returns the location of the arithmetic overflow vector. The arithmetic overflow flag is not enabled for CORTEX BASIC. To use arithmetic overflow a service routine should be provided.

The vector is one word which is the entry point for the routine. Return is via a RTWP instruction. The return context may be used to find the cause of the overflow.

4.3.5.19 SYS(17)

System function 17 returns the location of the illegal opcode vector. The standard routine simply prints 'ILLEGAL OPCODE' to the screen.

An alternative routine may be provided by setting the illegal opcode vector. This is one word pointing to the entry point of the routine. Return is via a RTWP instruction. R14 may be used to find the opcode that caused the interrupt.

4.3.5.20 SYS(18)

System function 18 returns the location of the print preprocessor vector. If set up, the preprocessor is called before output to the screen or any other device enabled by the UNIT statement. This allows any character translation or other operations to be performed on the print buffer.

The vector consists of two words, the first points to the routines workspace and the second is the entry point. Return is via a RTWP instruction. The start and end of the message to be printed are given by R7 and R8 respectively of the calling routine workspace. These may be found by indexing on R13 of the service routine workspace.

4.3.5.21 SYS(19)

System function 19 returns the location of the breakpoint 0 vector. As explained in sections 7 and 8, breakpoint number 0 has the extra feature of allowing a user routine to be called when the breakpoint is reached.

The vector is one word giving the entry point of the routine. Return is via an RT instruction.

4.3.6 Other Functions

4.3.6.1 COL

Form:

[line number] <variable> = COL(<exp1>,<exp2>)

The color function returns the color of the pixel at x,y coordinates of <exp1>,<exp2>. These are pixel coordinates as explained under the GRAPH statement. The color code is returned as an integer value. Color codes are listed under the COLOUR statement.

EXAMPLE:

```
A=COL(100,92)
```

A becomes equal to the color code for the pixel at 100,92.

4.3.6.2 RND

Form:

```
[line number] <variable> = RND
```

The random number function is used to generate a pseudo random number between 0 and 1. For example:

```
PRINT RND
```

might return a number like:

```
.2113190
```

To get a random number between 0 and a value, multiply RND by that value.

Example:

```
A=RND*10
```

returns a random number between 0 and 10.

The seed used for random number generation is set by the RANDOM statement.

4.3.6.3 TIC

Form:

```
[line number] <variable> = TIC(<expression>)
```

The time difference function samples the real time clock. It returns the current TIC value minus the expression value. For example:

```
T = TIC(0)
```

obtains the current time (in clock ticks, use TIME to get it in hours/minutes/seconds)

```
D = TIC(T)
```

calculates the elapsed time in ticks since we stored the time in T.

```
i.e.: TIC(T) is the same as TIC(0) - T
```

Each clock tick is 10 milliseconds, which is one hundredth of a second.

EXAMPLE:

```
10 W = RND*1000
20 WAIT W
40 T=TIC(0)
50 INPUT "PRESS RETURN" Z
60 R = TIC(T)
70 PRINT "YOU TOOK" ; R/100 ; "SECONDS"
```

This program is a reaction timer. First a random delay between 0 and 10 seconds is generated. Then the time is noted and a message output. When the user types return, the time delay is noted. The result is printed out.

5. THE VIDEO DISPLAY PROCESSOR

This section describes the video display processor (or "VDP" for short) used for graphics displays. Use this section in conjunction with section 4 which gives detailed syntax for the graphics statements and functions.

DISPLAY MODES

The CORTEX computer contains an advanced video processor (the TMS9928A or the TMS9929A) that has 16K bytes of dynamic memory dedicated to it. This memory is known video RAM (or "VRAM" for short) and is used to store all the information needed to generate complex graphic displays including character sets, shapes, color tables, etc. This 16K bytes of memory is completely separate from main program memory.

The VDP controls all access to the VRAM and also provides the refresh cycles needed by it. The TMS9995 accesses the VRAM via the memory-mapped VDP which appears as a single word in the CORTEX address space. As a result complex graphic displays do not take up large areas of main memory.

16 distinct colors are provided by the VDP under the control of the 'COLOUR' statement. These colors are:

transparent	light yellow	dark yellow	white
medium green	light green	dark green	grey
medium red	light red	dark red	black
magenta	light blue	dark blue	cyan

With CORTEX BASIC, the VDP is only used in either text mode or graphics 2 mode. Other modes are available to the assembly language programmer who should refer to the TMS9928A Data Sheet for full information. The VDP should always be returned to one of these 2 modes before returning to BASIC.

Note: Direct manipulation of the video display processor should not be attempted from a BASIC program as this will have unpredictable effects on the behavior of the system.

TEXT MODE

In this mode the TV screen has 24 lines of 40 characters that are displayed in the current foreground color with the backdrop set to the current background color. Scrolling is automatically performed when the screen becomes full.

The VDP is initialized in text mode when the computer is first switched on: you can return to this mode at any time by using the 'TEXT' statement.

A 'LIST' command will also force the VDP into text mode.

Note: No graphic operations are allowed in text mode.

GRAPHIC MODE

When graphic mode is entered (using the 'GRAPH' statement) you have the full use of CORTEX BASIC's graphic commands (e.g., 'PLOT', 'SHAPE', 'SPRITE', etc., as well as textual displays.

The display has 256 dots horizontally and 192 dots vertically: these dots are referred to as pixels and may take on any of the 16 available colors (with certain limitations horizontally which will be explained below).

The pixels that are set are in the foreground color and those that are not are in the background color. The foreground and background colors are set using the 'COLOUR' statement and may be changed to give multiple colors on the screen at any one time.

The SPRITE, PLOT and UNPLOT statements use pixel coordinates. Both X and Y coordinates range from 0 to 255. Since there are only 192 vertical screen positions, those from 192 to 255 are "off screen" and may be used for hiding predefined sprites.

This screen of 256x192 pixels is further divided into blocks of 8x8 pixels called cells. These are used by the SPUT and SGET statements. There are 32 of these cells horizontally and 24 vertically. Maps of the cells in TEXT and GRAPH modes are given in section 4 and Appendix F.

Within each of these cells there are two colors defined for each horizontal row of 8 pixels, one color (foreground color) for the pixels that have been set on and another color (background color) for those that have not. The pairs of colors for each pixel row within the cell may be set independently of the others.

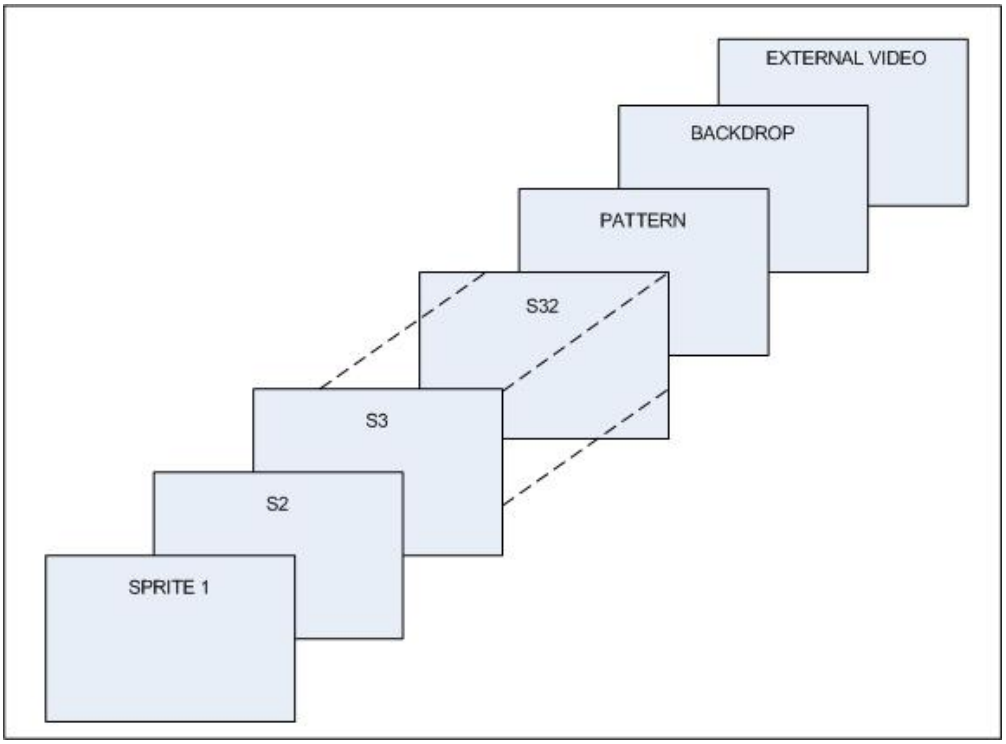
When printing text in graphics mode, use is made of the cells described above to hold characters. This means that there are only 32 characters available per line in graphics mode compared to the 40 in text mode. If printing past the bottom of the screen is attempted the screen will not scroll but will wrap around to the top of the screen.

As there is no visible cursor in this mode, the user should return to text mode via the 'TEXT' statement before editing a line, listing a program, etc.

The text characters are displayed in the current foreground color with the rest of the cell set to the current background color. The foreground and background colors may be changed during the printing of text to give more than one color of text on the screen at any one time.

In front of the pixel display are 32 other 'planes'. Each of these planes is capable of holding an object called a sprite. The top plane is numbered '0' and the one nearest the pixel display is numbered '31'. These planes are given priorities such that a sprite will conceal part of another sprite on a higher numbered plane and the pixel display if it overlaps them.

Apart from the sprite, the entire sprite plane is transparent. This feature allows the user to create a 3-D effect on the display.



Arrangement of video display planes

Each sprite can be set to any one of the 16 colors. Any pixels of the sprite that are not set are transparent and sprites underneath it will show through. If there are no sprites under it then the pixel display will show through.

The shape of the sprite is defined using the 'SHAPE' statement and its position and color on the screen are set using the 'SPRITE' statement. The 'MAG' statement is used to define the size of the sprite and how many shape table entries are used to form the sprite shape.

Up to 32 sprites may be active at one time. However, on any one horizontal pixel line, only 4 may be visible. If more than 4 sprites are active together on a line, the one of lowest priority (i.e.: nearest the backdrop) will be invisible.

Note: Sprites are only available in graphic mode.

Shapes defined in the shape table may also be placed onto the pixel display using the 'SPUT' statement. Thus a sprite that will not move for some time may be transferred onto the backdrop to free the sprite plane for other use.

The 'SGET' statement is used to pull a shape from the screen into the shape table. Each bit that is 'on' (i.e.: in the foreground color for that group of 8 pixels) is set to 1 in the shape table entry.

NOTE: 'SPUT' and 'SGET' have different effects in text mode, see section 4.

In addition to the 32 sprite planes, pixel display and backdrop there is one further plane behind all the others. This is known as the external video plane.

The hardware for using external video is not implemented in the CORTEX. To add this feature the external video should be mixed externally to the 9928 and gated with the color difference outputs. These outputs indicate when external video should be active. The TMS9900 family data book gives further details of how to do this.

If external video has been implemented, this plane appears behind all the others. It will only show through where all other planes, including the backdrop, are set to transparent. In the absence of external video, the external video plane will appear black.

This page is intentionally blank

6. CHARACTER STRINGS

6.1 General

This section explains the use of character strings in CORTEX BASIC. The methods used are different from those found on many microcomputer systems. This approach has been adopted to obtain the fastest possible execution speed. Once the differences have been mastered, string handling on the CORTEX is very powerful.

A character string is a group of ASCII characters, for example: "ABCDE". Literal strings of this sort are enclosed in quotes, ' or ". The same type of quote must end the string as started it. Thus: 'ABC"DE' is a valid string, the " character being part of the string.

6.2 String Variables

Strings may be stored in BASIC variables. When a variable is being used to store a string it is preceded by a $ sign. A string variable may be assigned a value by the LET statement:

```
LET $A="ABCD"
LET $B=$A
```

or (since LET is optional):

```
$A="ABCD"
$B=$A
```

Note that $A is the same variable as A. The $ indicates that it is being used to store a string.

Non printable characters may be included in a string by enclosing the hexadecimal ASCII code in angle brackets.

```
$A="A<0A>"
```

This string will print as "A" followed by a line feed (character code 0A).

Each character stored takes one byte of memory. A simple undimensioned variable can store up to 5 characters. The variable is six bytes long and the last byte is used to store a null terminator. Care should be taken not to exceed this as other variables will become corrupted.

6.3 Longer Strings

Very often it is necessary to store strings longer than five characters. This is done using array variables. Each array element can store six characters, except the last which holds five and the terminator. Thus the maximum length of a string is (6*number of elements)-1. Note that elements start at zero, so an array dimensioned to 9 has 10 elements. The normal DIMension statement is used:

```
DIM $A(20)
```

This array will hold up to 125 characters (i.e.: 6*21 -1). Since A(x) and $A(x) refer to the same variable the $ sign in the DIM statement is optional and is in fact ignored by BASIC. It is advisable to include it for variables to be used for strings for your own reference.

To use the array as a character string, refer to element zero. For example:

```
$A(0)="Hello this is a character string"

PRINT $A(0)

Hello this is a character string
```

Note that although A and $A are the same variable, $A and $A(0) are distinct. $A can only hold 5 characters. $A(0) as dimensioned above can hold up to 125. Although allowed, it is best not to use an undimensioned variable and array of the same name in a program as this can cause confusion to the programmer.

6.4 Arrays of Strings

To generate an array of strings simply DIMension the variable to one more dimension than required. The extra dimension gives the length of the strings. For example:

```
DIM $B(10,10,5)
```

generates a 10 by 10 array of strings, each holding up to 35 characters.

The last element defines the length of the strings. The elements of the string array are then referred to as:

```
$B(X,Y,0)="String at array position X,Y"
```

6.5 String Comparisons

For the rest of this section the following conventions will be used:

<$var> means either a literal string in quotes or a variable name preceded by a $ sign.

$<var> means only a $ sign preceding a variable.

Comparisons of the following form are valid:

```
IF <$var> <relation> <$var> THEN <BASIC statement>
```

Examples:

```
100 IF $IP = "Y" THEN GOTO 500
110 IF $N(I,0) > $N(J,0) THEN GOSUB 600
```

Any of the relations, = <> > < <= >= may be used.

Comparisons are made according to the ASCII codes of the strings. Thus "B" is greater than "A", "FFGA" is greater than "FFG" and so on.

6.6 Reading Strings

Strings may be read from a DATA statement using a READ statement provided that the string data is read into a string variable.

Examples:

```
10 DIM $N(5),$Z(5)
20 READ $N(0),A,B,$Z(0)
30 DATA "STRING DATA" , 12345 , A*10 , $N(0)
```

In this example, $N(0) receives the string "STRING DATA", the variable A receives the number 12345 and B the number 123450. The string variable $Z(0) receives the same string as $N(0).

6.7 Indexing into Strings

A dimensioned string variable can have an offset into the string. This is achieved by following the last subscript with a semicolon and the character displacement. The range of the index is from one to the length of the string. $A(0;1) is the same as $A(0).

Examples:

```
10 DIM $A(10)
20 $A(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 PRINT $A(0)
40 PRINT $A(0;1)
50 PRINT $A(0;10)
60 STOP

RUN

ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
JKLMNOPQRSTUVWXYZ

Stop at 60

10 DIM $A(10),$B(10)
20 $A(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 $B(0)=$A(0;10)
40 $A(0;2)=$B(0;2)
50 PRINT $A(0),$B(0)
60 STOP

RUN

AKLMNOPQRSTUVWXYZ      JKLMOPQRSTUVWXYZ
```

6.8 String Concatenation

Strings are concatenated using the "+" operator.

Form:

```
$<var> = <$var> + <$var> + ...
```

A number of concatenations may be chained together.

Example:

```
10 DIM $A(10),$B(10)
20 $A(0)="ABCDE"
30 $B(0)=$A(0)+"FG"+"HIJK"
40 PRINT $B(0)
50 STOP

RUN

ABCDEFGHIJK

STOP AT 50
```

6.9 Character Pick

Characters can be picked from one variable into another by using the semicolon as an offset and a comma to indicate the number of characters.

Form:

```
$<var> = <$var) , <expression>
```

The expression gives the number of characters to be used. The characters specified are transferred to the new variable. If an offset is specified in the variable to receive the characters, they are copied in starting at that offset. The new string terminates at the end of the picked characters.

Example:

```
10 DIM $A(10),$B(10)
20 $A(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 $B(0)=$A(0;4),6
40 $B(0;5)=$A(0),1
50 PRINT $B(0)
60 STOP

RUN

DEFGA

Stop at 60
```

If the number of characters to be picked goes beyond the end of the source string, then characters up to the end of the string are picked. If the expression is negative, no characters are picked.

6.10 Character Replacement

Character replacement is very similar to character pick. In this case the new string retains its original length. It does not terminate at the end of characters picked.

Form:

```
$<var> = <$var> ; <expression>
```

The expression gives the number of characters to be replaced. The characters specified are transferred to the new variable. If an offset is specified in the variable to receive the characters, they are copied in starting at that offset. The new string retains its original length or is extended if the new characters make it longer.

Example:

```
10 DIM $A(10),$B(10)
20 $A(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 $B(0)=$A(0;4),6
40 $B(0;5)=$A(0);1
50 PRINT $B(0)
60 STOP

RUN

DEFGAI

Stop at 60
```

If the number of characters to be replaced goes beyond the end of the source string, then characters up to the end of the string are taken. The new string will end at the end of the characters replaced. If the expression is negative, no characters are picked.

6.11 Individual Character Replacement

Individual characters may be replaced by using the ASCII character code of the new character preceded by the "%" operator.

Form:

```
$<var> = %<expression> ...
```

Example:

```
10 DIM $A(10),$B(10)
20 $A(0)="**********"
30 $A(0;3)=%65%66
40 $B(0)=%65%66%0
50 PRINT $A(0),$B(0)
60 STOP

RUN

**AB*******        AB

Stop at 60
```

If the replacement goes beyond the end of the existing string, you must put the null terminator in yourself as in line 40 above. With the "%" operator this 1s NOT automatic. For this reason, care is needed to use the character replacement operator.

6.12 Character Insertion

Characters can be inserted into a string variable using the "/" operator.

Form:

```
$<var> = / <$var>
```

The characters will be inserted at the beginning of the string. If an offset is specified, the characters will be inserted before the specified character.

Example:

```
10 DIM $A(10),$B(10)
20 $A(0)="ABCDEFG"
30 $A(0;4)=/ "...."
40 PRINT $A(0)
50 STOP

RUN

ABC....DEFG

Stop at 50
```

If the offset is greater than the length of the variable, no insertion takes place.

6.13 Character Deletion

Characters are deleted from a string variable by using the "/" operator followed by an expression.

Form:

$<var> = /<expression>

The expression is evaluated and that number of characters are deleted from the beginning of the string. If an offset is specified, then characters are deleted starting with the specified character.

Example:

```
10 DIM $A(10),$B(10)
20 $A(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 $A(0;5)=/10
40 PRINT $A(0)
50 STOP

RUN

ABCDOPQRSTUVWXYZ

Stop at 50
```

6.14 Convert String to Number

A character string may contain characters that make up a valid number. This number may be given to a numeric variable by assigning the string to the numeric variable.

Form:

```
<var> = <$var>,<var>
```

The first character in the string that is not a valid numeric digit is placed in the second variable. If this is a null then all characters have been successfully converted. This error variable may be used as a numeric variable, as it is specified in the conversion, and checked for "0" value. Alternatively, a "$" is placed in front and it is used as a string variable. It may then be printed or tested for the invalid character.

Example:

```
10 N="1234",E
20 N1="12DE",E1
30 PRINT N,$E
40 PRINT N1,$E1
50 STOP

RUN

1234
12          D

Stop at 50
```

6.15 Convert Number to String

A number can be converted to a string by assigning the number to a string variable. The string will contain the characters that would be printed by printing the number.

Form:

```
$<var> = <expression>
```

Example:

```
10 DIM $A(10),$B(10)
20 N=1
30 $A(0)=4*ATN(N)
40 $B(0)=SQR(2)
50 PRINT $A(0),$B(0)
60 STOP

RUN

3.141592        1.414213

Stop at 50
```

Formatted conversions may be made by preceding the expression with "#" and a string. The formatting rules are the same as those used for the PRINT statement (section 4.2.7).

Example:

```
10 DIM $A(10),$B(10)
20 $A(0)=#"999,990.99",5465
30 $B(0)=#"<<<,<<<.00>",-5465
40 PRINT $A(0),$B(0)
50 STOP

RUN

  5,465.00           <5,465.00>

Stop at 50
```

6.16 Character Functions

In addition to all the operations explained in this section, there are a number of standard string functions. These are explained in detail in section 4.3.2, String Functions.

The functions available are:

```
ASC(<$var>)          - Convert string to ASCII code
LEN(<$var>)          - String length function
MCH(<$var>,<$var>)   - Match between two strings
POS(<$var>,<$var>)   - Position of 1 string within another
```

Each function returns a numeric value.

This page is intentionally blank

7. MONITOR

7.1 Introduction

In addition to BASIC, the CORTEX is supplied with an assembly language and machine code monitor. This may be used to write programs in TMS9995 assembly language. These may either be executed in their own right or CALLed from BASIC.

The CORTEX monitor is a powerful tool to simplify the debug of assembly language programs. To achieve this, there are several modes of single stepping, a 'Trace' facility, an assembler, a disassembler and multiple breakpoints.

All commands are single letter commands, followed by up to three Hex parameters.

The monitor is entered by typing the MON command in BASIC. The message:

```
* Monitor Rev. x.x (C) 1982
```

is output. The number x.x gives the version of the monitor. The prompt [] is displayed. Return to BASIC is by typing "C". The monitor is now ready to accept commands.

NOTE: In this section, assembly language is used to mean the source mnemonics for TMS9995 code. Machine code is the actual code generated for execution by the 9995, here represented by hexadecimal numbers.

7.2 Monitor Commands

7.2.1 List of Commands

The commands available in the monitor are as follows:

?	WHERE AM I
A	LINE BY LINE ASSEMBLER
B	BREAKPOINT SETTING
C	CRU INSPECT CHANGE
D	DUMP TO TAPE
E	EXECUTE
F	FIND WORD OR BYTE
G	GOTO BASIC (WARM START)
I	INITIALIZE MEMORY
L	LOAD FROM TAPE
M	MEMORY INSPECT CHANGE
N	NEGATIVE FIND (FIND ANYTHING BUT PATTERN)
P	OUTPUT PORT ENABLE/DISABLE
R	INSPECT/CHANGE WP, PC, AND ST REGISTERS
S	SINGLE/MULTIPLE STEPPING
T	TRACE-SINGLE STEP WITH PRINTOUT
U	UN-ASSEMBLER
W	WORKSPACE REGISTER INSPECT/CHANGE
X	TRANSFER (XFER) MEMORY

7.2.2 Notes On Command Format

The following notes are given mainly for those users who have not used a debug monitor previously. The formats given for command input under each command description, whilst sufficient for those who are familiar with monitors, do not give the full formats required.

1. Items enclosed by '<>' are user supplied, items enclosed by '[]' are monitor supplied.

2. Valid terminators etc., are:

   a. Carriage Return

   b. Space

   c. Comma

   d. Minus sign

A Carriage Return is used as a command parameter input terminator.
A Space is used as a parameter delimiter and a "stepper".
A Comma is used as a parameter delimiter only.
A Minus sign is used as a "negative stepper".

All numbers are hexadecimal and are entered as up to 4 hexadecimal digits. Do NOT type H after the number.

Any deviations from these are given in the relevant commands.

DELIMITERS ARE REQUIRED BETWEEN USER INPUT PARAMETERS

7.2.3 Memory Data Checking

NO checking is carried out in the "M" and "L" commands to ensure that the data which appears in memory is that which was written. The two main reasons for this are:

1. Problems occur with memory mapped ports of various types (particularly those which auto-increment on a read or write) if a write to memory is always followed by a read.

2. When a system is configured such that RAM is overlaid with EPROM (quite common with 64K DRAM systems) the RAM can be written to but if a check is carried out by reading the data back the EPROM will be read which will (usually) give a write error.

NOTE

All commands may be aborted while in the user input mode by pressing ESCAPE.

7.3 Monitor Command Descriptions

7.3.1 ? = Where Am I?

The current values of WP,PC and ST are printed together with the instruction about to be executed at the current PC.

7.3.2 A = Line By Line (Zero Label) Assembler

The A command is followed by the assembly address. 9900 series assembly language mnemonics are entered line by line until the escape key is pressed.

NOTE

The format of TMS9995 instructions follows the standard 9900 series assembler syntax This means that hex numbers must be preceded by the > character.

* Starting Format:
  - A <ADDRESS><TERMINATOR>
* Continuing Format:
  - [ADDRESS][spaces]<OPCODE><space><OPERANDS><RETURN>
* Response:
  - [ADDRESS][OBJECT]
  - [ADDRESS][spaces]<OPCODE><space><OPERANDS><RETURN>

* Assembler default settings.

  - Address = start of user RAM area (Can be changed at assembly time)

* To return to the command scanner, use ESCAPE.

NOTE

The user RAM area is that area between the BASIC interpreter and the bottom of the BASIC programs. The top of this area is defined by the NEW command. This area allows assembly language routines to reside co-resident with BASIC programs without being corrupted by BASIC. Alternatively, assembly language routines may be placed within array variables.

The opcode must be separated from its operand by a space character. Comments may be added if required BUT they must be separated from the operand by at least one space character (comments have no effect on the object i.e., they are not stored).

Error checking is done wherever possible, but in some cases errors are not detected by the assembler (e.g., DATA +2 will always result in >0000 being output). Because of this it is good practice to always disassemble the object code produced (using the 'U' command) in order to ensure that the assembled code is correct. Errors are indicated by one of the following:

*S = Syntax error
*D = Data (number) error
*R = Register error

This assembler is capable of assembling all the 9900/9995 opcodes (including the pseudo opcodes RT, NOP and SPIN) and the DATA directive. The predefined macro instructions can be assembled using the formats given in section 8.

7.3.2.1 Expressions

Only '+' and '-' are allowed as mathematical expressions.

7.3.2.2 Constants

All constants are assumed to be decimal unless prefixed by a character to indicate otherwise:

```
Decimal   1234
Hex       >1234
Binary    %110111100
ASCII    'AB' (=>4142)
```

Note that 'A' gives >0041 and not >4100

7.3.2.3 Program Counter Relative

Program counter relative assembly (for Jumps) has the normal format e.g., JMP $-2 will result in >10FE as the object output.

7.3.2.4 Text (and Unprintable Characters)

Text is preceded by a '$' and can be entered as follows:

```
$THIS IS AN EXAMPLE OF TEXT
```

and will result in the appropriate hex code being entered into memory. If the last character is on an 'odd' boundary then a SPACE character will be added to force the address to an even value. Unprintable characters may be entered in this mode with the exception of carriage return. However remember that the character is echoed to the terminal, and therefore cursor controls will affect the cursor position.

7.3.2.5 Assembly Address Change

To change the assembly address simply enter a '/' followed by the address required and a terminator.

7.3.2.6 Peculiarities

The assembler will accept a number preceded by a minus sign. If negative numbers are required, they should be entered as Hex words, i.e., -1 = >FFFF

7.3.3 B = Breakpoint Inspect/Change

The user is allowed up to 16 breakpoints. Instructions at designated breakpoints are replaced by a BKPT macro instruction during the Execute command and replaced on reentry to the monitor's command scanner (e.g., RESET is pressed to regain control, a breakpoint is reached etc.)

* Format:

- B <CARRIAGE RETURN> Entering just B will display the current settings of the 16 breakpoints.

- B <BREAKPOINT NUMBER><TERMINATOR> Entering the breakpoint number after B will display the current address of that breakpoint and wait for the new address to be entered.

- B <-> The B- command clears all breakpoints.

All of the 16 breakpoints whose value is not zero will be set in the Execute command (if required). Breakpoints are invisible to the user under normal circumstances (that is, the user should never see the breakpoint instruction in the program). The situation can however occur whereby the user program causes the processor to corrupt the Monitor flags. If this occurs, the breakpoints should be cleared and the breakpoints removed from RAM by use of the 'M' command and replaced with the appropriate instructions. Breakpoints are not removed from the breakpoint list when 'hit' during execution. If execution of the user program commences from a point at which a breakpoint is to be set then the instruction at that point is executed before the breakpoint is inserted into the program.

NOTE

Breakpoints will be removed from the program even if no Breakpoint has been reached. This can happen when the user program returns to the Monitor command scanner or 'Reset' is pressed to regain control.

When a breakpoint is reached, the breakpoint number is output, followed by the processor status as described in 'Trace'. Control is then returned to the command scanner.

7.3.3.1 Breakpoint Zero

This breakpoint has, in addition to its normal use, a special facility associated with it in the form of a user subroutine capability. When breakpoint zero is reached, the breakpoint zero vector is checked to see whether or not it is zero. If it is not zero, then the data there is used as the address to branch to for the user subroutine. The address of this vector is found via the SYS(19) function in BASIC. The user subroutine is accessed via a 'branch and link'. If the user subroutine decides that the conditions required have not been met then the breakpoint can be ignored (on this occasion) by returning to the monitor by means of an 'RT' instruction.

If the flag is set to zero, then breakpoint zero works in the same manner as the remaining breakpoints.

7.3.4 C = CRU Inspect/change

Input the CRU base address followed by the bit count. All input and output to the CRU is right justified in the 16 bit input/output data fields.

Input of a Carriage Return as a termination character returns control to the command scanner. A space as termination character causes the CRU input bits to be set, and the (changed) CRU output bits to be re-output to the terminal.

* Format:
  - C <BASE ADDRESS)<No. OF BITS><TERMINATOR>
  - C <BASE ADDRESS><TERMINATOR>
* Defaults:
  - Base = 0
  - No. of bits = 16

7.3.5 D = Memory Dump to Cassette

Dump memory image to the audio cassette interface. The data is output in absolute code. This means that it can only be reloaded to the location from which it came when loaded by the 'L' command.

The dump procedure is very similar to that used for SAVE in BASIC. The D command produces a prompt for IDT. This corresponds to the name in BASIC. The Auto Run and cassette ready prompts follow.

* Format:
  - D <START ADR><STOP ADR><ENTRY ADR><TERMINATOR>
  - D <START ADR><STOP ADR><TERMINATOR>
* Followed by:
  - [IDT=]<up to 8 chars>
  - [Auto Run? (Y/N)] <Y or N>
  - [Cassette ready? (Y/N)) <Y or N>
* Defaults:
  - All address defaults are 0
  - IDT = 8 spaces
  - READY = N(o)

Memory is dumped in the following format:

TAPE SAVE FORMAT

SYNC CHAR (16H)	Repeated for 2 second startup
STX CHAR (02H)	Start of data
HEADER BLOCK	See below
MEMORY IMAGE	
ETX CHAR (03H)	End of data
CHECKSUM	Checksum of memory image

HEADER BLOCK

AUTO RUN FLAG	RUN =>A5A5 NORUN=>5A5A
8 BYTE NAME	NULL FILLED
>0000	
LOAD ADDRESS	
ENTRY POINT	(OFFSET FROM LOAD ADDRESS)
LOAD LENGTH	(BYTES)
CHECKSUM	HEADER ONLY

7.3.6 E = Execute

Ask the user if the defined breakpoints are to be set, then output the present WP,PC & ST and allow the user to alter the PC if required. On input of a terminator, execute the user program.

* Format:

  - E [Set BPs?] <Y or N> [Execute WP=XXXX PC=XXXX ST=XXXX]<PC><TERMINATOR>

  - E [Set BPS?] <Y or N> [Execute WP=XXXX PC=XXXX ST=XXXX]<TERMINATOR>

* Defaults:

  - WP,PC,ST = as given

NOTE

If the WP or ST need to be changed, the 'R' command should be used. To allow the system clock to continue running and to enable break in to the program with the escape key, interrupts must not be disabled. To achieve this, load the status register with 0F before executing.

7.3.7 F = Find Word or Byte

Look from the Start address to the Stop address for the specified data pattern. If the pattern is found, then output the address. The termination character determines the search mode:

```
Carriage Return = WORD SEARCH
Minus sign      = BYTE SEARCH
```

* Format:

  - F <START ADR.><STOP ADR.><PATTERN><TERMINATOR>

* Defaults: All zero

See also 'N' command (Negative find)

7.3.8 G = Goto BASIC

The G command returns to BASIC by doing a warm start into the interpreter. BASIC programs previously entered will still be intact provided they have not been overwritten by commands or programs executed from the monitor.

7.3.9 I = Initialize (Pattern) Memory

Initialize memory from Start address to Stop address with a given pattern.

* Format:
  - I <START ADR><STOP ADR><PATTERN><TERMINATOR>
* Defaults: All zero

7.3.10 L = Load Memory From Cassette

Accept the IDT from the user. Turn on cassette motor and load the data. Upon a good load, control is passed to the calling routine (normally the monitor command scanner). A checksum error produces the message:

** Tape read error **

The (possibly corrupted) data will still have been loaded.

* Format:
  - L <TERMINATOR>
* Followed by:
  - [IDT=]<up to 8 chars>
  - [Cassette ready? (Y/N)] <Y or N>

7.3.11 M = Inspect/Change Memory

There are two options for this command:

1. Output the memory location followed by its contents and allow the contents to be changed.

   * Format:

     - M <ADDRESS><CARRIAGE RETURN>

   * Response:

     - [ADDRESS=CONTENTS]<NEW DATA><TERMINATOR>

     - [ADDRRSS=CONTENTS]<TERMINATOR>

   * The terminator determines what is to be done next:

     - CARRIAGE RETURN = Return to the command scanner

     - SPACE = display next address for change

     - MINUS = display previous address for change

NOTE

If new data is entered, then this will be placed in memory regardless of the terminator. Leading zeros are not required for the entry of data. If a mistake is made during entry then simply enter the correct four hex digits (the monitor takes only the last four hex digits entered).

2. Dump the memory contents to the terminal from the start address to the stop address.

   * Format:

     - M <START ADR><STOP ADR><CARRIAGE RETURN>

   * Defaults: Zero

7.3.12 N = Negative 'Find'

Search memory from the Start address to the Stop address for patterns other than the specified data pattern.

If a data pattern other than the one specified is found then the address of that pattern is output. The termination character determines the search mode.

```
Carriage Return = WORD SEARCH
Minus sign      = BYTE SEARCH
```

* Format:

  - N <START ADR><STOP ADR><PATTERN><TERMINATOR>

* Defaults: zero

7.3.13 P = Port Toggle

Load the unit flags with the parameter.

* Format: P <NEW UNIT FLAG><TERMINATOR>

This command allows devices to be enabled and disabled from the monitor. The unit flags area is a 16 bit word, each bit set indicates a device enabled. See the UNIT statement and SYS(5) functions in section 4 for further information about devices and device numbers.

7.3.14 R = Inspect/Change WP, FC, ST Registers

Allow the user to inspect/change the Workspace Pointer, Program Counter and Status Registers.

* Format:

  - R <CARRIAGE RETURN>

* RESPONSE:

  - [REG = CONTENTS]<NEW VALUE><TERMINATOR>

* Terminators:

  - SPACE = To next register

  - MINUS = To previous register

  - RETURN = To command scanner

* Defaults: NONE

7.3.15 S = Single Step

This command has three different modes of operation:

1. Execute the specified number of single steps.

    * Format:

        - S <VALUE LESS THAN HEX 80> <TERMINATOR>

The processor will execute the required number of single steps and then return to the command scanner via a print routine (see below).

2. Single step until the program counter reaches the given address.

    * Format:

        - S <VALUE GREATER THAN HEX 80><TERMINATOR>

The processor will execute the routine (as indicated by the 'R' command) until the Program Counter reaches the required address. At this point, control is passed back to the command scanner via a print routine (see below). This mode of single step effectively gives a 'Breakpoint in EPROM' facility.

3. Single step until the contents of location "ADDRESS" equals VALUE.

    * Format:

        - S <ADDRESS> <VALUE>

The processor will single step as stated above. Upon the address containing the required value, control is passed back to the command scanner via a print routine which gives the following information:

```
[SS: WP=XXXX PC=XXXX ST=XXXX DATA @ PC DISASSEMBLY OF PC DATA]
```

NOTE

> In modes one and two of single step the data given is for the instruction ABOUT to be executed and not the last instruction executed. In mode three the previous instruction is the one which caused ADDRESS to contain VALUE.

* Defaults:

  - Steps = 1

  - Addresses = 0

NOTE

Real time execution cannot be obtained in single step mode, (execution will take at least 10 times longer than real time). See also the notes in section 7.3.19 about stepping through MIDs.

7.3.16 T = Trace

Trace the path of the processor through the required routine, returning to the print routine (as in Single step above) after each instruction.

* Format:

  - T <NUMBER OF STEPS> <TERMINATOR>

* Default: 1 step

7.3.16.1 Tracing and Single Stepping through MIDs

Due to the print routines called by 'Trace' and 'Single Step' using the pre-defined MIDs, a problem would arise if an attempt was made to trace or step into these, as the data in R13, R14 & R15 would be overwritten by these calling routines. In order to avoid this problem, the Trace and Single Step print routine is not allowed to be called during these MIDs, and the single step will continue until the MID is complete. The effect of this to the user is that a MID will appear to be only one instruction. e.g., if the next instruction to be executed is the MSG MID and one single step is done, then the whole message will be output before control returns to the command scanner.

7.3.17 U = Un-Assembler/Disassembler

There are two modes of this command:

1. Disassemble a line at a time from the given address:

    * Format:

      - U <ADDRESS> <TERMINATOR>

    * response:

      - [DISASSEMBLY] <TERMINATOR> etc., for each line

Press ESCAPE to return to the command scanner.

    * Default: Start of user RAM

2. Disassemble block from Start address to Stop address:

    * Format:

      - U <START ADDRESS><STOP ADDRESS><TERMINATOR>

    * Defaults: zero

The disassembler is also called by the 'Breakpoint', 'Single Step' and 'Trace' routines.

7.3.18 W = Inspect/Change User Workspace Register

The W command has two modes of operation:

1. Display the contents of all the current user workspace registers and return to the command scanner.

   * Format:

     - W <CARRIAGE RETURN>

2. Input the register number in Hex, and display the contents of the register for change. The output is handled in a similar way to memory (M) inspect/change mode.

   * Format:

     - W <REGISTER NUMBER><TERMINATOR>

   * Default:

     - Not applicable (Mode 2 defaults to mode 1)

7.3.19 X = Transfer (Xfer) Data From One Block of Memory to Another

Copy the data in block from (Start address to Stop address) to the new address.

   * Format:

     - R <START ADR><STOP ADR><NEW START ADR><TERM>

   * Defaults: zero

NOTE

Data transfers are carried out in 'Byte' mode and are always done in such a way as not to overwrite data which has yet to be moved. This command is useful for such things as moving data up down by a word in order to insert/delete an instruction and also for transferring data from EPROM to RAM. NOTE also that the two start addresses are allowed to be equal thus permitting transfer of data to RAM which is overlaid with EPROM.

7.4 Programming Notes

7.4.1 Choice of Memory Locations

For a program to be CALLed from a BASIC program, no parts of BASIC or its data storage should be overwritten. The NEW command from BASIC may be used to reserve memory for the machine code routine. Alternatively, space may be reserved in a array using the DIM statement. The address allocated to the array may be found using the ADR function.

If a program is to stand alone as machine code, it may overwrite parts of the CORTEX software. If it is desired to retain the monitor for debugging and to make use of existing device drivers and macro instructions, the program should be loaded above 2000 Hex. It should also be noted that the monitor cannot be used to single step or trace through code below 6000 hex. Memory above EC00 Hex should not be used as this is where system flags and workspaces are stored.

This means that for a program of this sort, 51K bytes between 2000 and EC00 Hex are available.

A program that provides its own interrupt service routines, device drivers, macroinstructions, debugging etc., may overwrite everything and take over the Cortex completely. An example of this would be UCSD Pascal booted from disc. 60K bytes of memory is available from zero to F000 Hex with TMS9995 internal RAM from F000 to F0FB and FFFC to FFFF Hex. The remaining 3.75K bytes of the map is used for memory mapped I/O. Full memory and I/O maps are given in Appendix E.

For all these types of program, further memory may be made available by using the memory mapper chip to access extra memory through the E-bus.

7.4.2 Ending Machine Code Programs

A program CALLed from BASIC should return using the RTWP instruction. Ensure that the program does not corrupt its return context.

A stand alone program may return to the monitor by using the instruction:

```
B @>80
```

This returns to the [] prompt.

A return to BASIC may be made by branching to the location returned by the SYS(12) function in BASIC.

7.4.3 Interrupt Mask

To enable all existing device drivers and keep the clock going while a machine code routine is executing, interrupts should not be disabled. To ensure this, the status register should be loaded with a value of 0F Hex or greater before execution. This also allows the program to be halted using the escape key.

8. MACRO INSTRUCTIONS

The CORTEX software contains a number of predefined instructions for I/O operations. These are serviced by the TMS9995 Macro Instruction Detect (MID) mechanism and are known as MIDs. Input and output takes place from and to all devices currently enabled in the unit flag. This is set up by the 'P' command or the UNIT statement in BASIC.

8.1 Pre-defined MIDs

The MIDs that are defined on the CORTEX for I/O operations are as follows:

```
BKPT  - Breakpoint instruction (use ONLY via the 'B' command)
MSG   - Output a message
READ  - Read a character
WRIT  - Write a character
EKO   - Read a character and echo it
WHXW  - Write Hex word
RHXW  - Read Hex word
WNBL  - Write Hex nibble
```

These opcodes are accepted by the line by line assembler and recognized by the disassembler.

8.2 MID Calling Sequences

8.2.1 Breakpoint: BKPT

- Call = BKPT Breakpoint number

- Return = Monitor Command Scanner

- MID opcode = 0FC0

CAUTION

THIS MID SHOULD NOT BE CALLED EXCEPT BY USING THE 'B' COMMAND.

8.2.2 Message Output: MSG

- Call = MSG (source address)

- Return = next instruction

- MID opcode = 0F80

The message pointed to by the source address is output until a zero byte is encountered. Return is to the caller.

Example:

```
MSG @>6200
```

Where the message is stored from >6200 upwards.

The output is byte by byte and uses the WRIT MID.

8.2.3 Read ASCII Character: READ

- Call = READ (destination address)

- Return = next instruction

- MID opcode = 0F40

The CORTEX waits for a character to be received from the keyboard. The destination address is cleared and the input character is placed in the LEFT HAND byte (MSB) of this word. The destination address should be on a word boundary, which means it must be an even number.

Example:

```
READ R5
```

Where the input character will be stored in R5.

8.2.4 Write ASCII Character: WRIT

- Call = WRIT (source address)

- Return = next instruction

- MID opcode = 0F00

The character pointed to by the source address is output.

Example:

```
WRIT R5
```

The character in the MSB of R5 will be output.

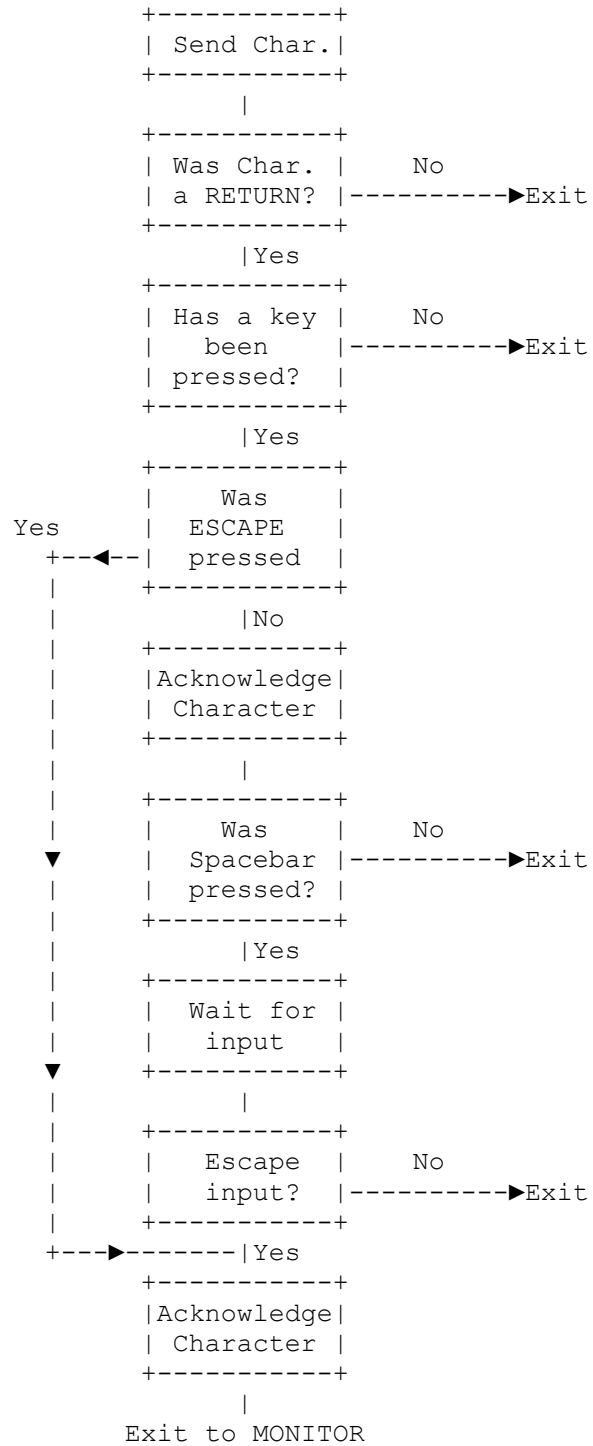
As can be seen from the flow diagram, each time a carriage return is output, a check is made to see if a character has been received. If one has and it is a Space, then output is halted. Action then is determined by the character:

Escape: will terminate output and return to BASIC or the Monitor.

Space: Wait for another character to be input, but do not acknowledge it. If character is other than a space, then next time a carriage return is output it will be acknowledged and ignored. If the character is a space then output will again be halted.

The above gives the user the capability of single stepping lines being output by means of the space bar, aborting by means of Escape, or going back to normal mode by pressing any other key.

The following flow diagram illustrates execution of the WRIT instruction:



8.2.5 Echo ASCII Character: EKO

- Call = EKO (destination address)
- Return = next instruction
- MID opcode = 0EC0

This MID calls the READ and WRITE MIDs. The destination address must be on a word boundary (normally a register).

Example:

```
EKO R6
```

This inputs a character into R6 and echoes it to the screen.

8.2.6 Write Hex Word: WHXW

- Call = WHXW (source address)
- Return = next instruction
- MID opcode = 0E80

The hex word at the source address is converted to four ASCII characters. The WRIT MID is used to output them to the screen. The source data is not affected.

Example:

```
WHXW *R5
```

This prints in Hex the contents of the word pointed to by R5.

8.2.7 Read Hex Word: RHXW

- Call = RHXW (destination address)
- Null return address = DATA
- Error return address = DATA
- Normal return = next instruction
- MID opcode = 0E40

Characters are input until a non Hex character or a termination character is found. (Termination characters are: space, comma, minus or carriage return.) The termination character is returned in the left hand byte of the word following the destination address, with the right hand byte set to zero.

Having input Hex characters and received a valid terminator, the Hex number is returned right justified in the destination address. Return is to the address following the Error Data statement.

If ONLY a termination character is found, then return is to the Null return address pointed to by the DATA statement following the call. The destination address remains unchanged. (This allows default values to be set up in the destination address before calling this MID).

If a non Hex character which is not a terminator is input, then the return is to the Error return pointed to by the DATA statement following the Null return pointer. The destination address and the word following it will remain unchanged.

Example:

```
RHXW R4
DATA >7000
DATA >7010
```

This will read a Hex word into R4 and the terminator into the MSB of R5. Normal return is to the instruction following the second data statement. If no characters are input before the terminator, return is to >7000. If invalid characters are input, return is to >7010.

8.2.8 Write Hex Nibble: WNBL

- Call = WNBL (source address)
- Return = next instruction
- MID opcode = 0E00

The right most Hex digit in the source address is converted to ASCII and output to the terminal using the WRIT MID. The source data is unaffected.

```
WNBL @>8000
```

This will output the ASCII character corresponding to the least significant Hex digit in the word at >8000.

Any other MID opcodes will produce a message:

```
** Illegal opcode **
```

and return into BASIC.

This page is intentionally blank

APPENDIX A

Alphabetical list of BASIC keywords

COMMANDS :-

DELIMITERS :-

TO TAB STEP THEN : , ; ? " ' [ ] ( )

FUNCTIONS :-

SYSTEM FUNCTIONS :-

APPENDIX B

List of BASIC keywords by type

PROGRAM GENERATION		PAGE
NEW	START A NEW PROGRAM	4-4
SIZE	PRINTS THE AMOUNT OF AVAILABLE MEMORY	4-5
NUMBER	AUTOMATIC LINE NUMBERING	4-11
ERROR	ENABLES/DISABLES ERROR LINE EDITING	4-17
RENUM	RENUMBERS THE PROGRAM	4-12
LIST	LISTS ALL OR PART OF THE PROGRAM	4-9
PURGE	DELETES SECTIONS OF THE PROGRAM	4-14
SAVE	SAVES THE PROGRAM ON CASSETTE	4-7
LOAD	LOADS A NAMED PROGRAM FROM CASSETTE	4-6
RUN	EXECUTES THE CURRENT BASIC PROGRAM	4-10
CONT	RESUMES EXECUTION AFTER HALTING	4-10
ENTER	ENTER STRING AS A PROGRAM LINE	4-89

COMMENTS AND DECLARATIONS		
REM	COMMENT, HAS NO EFFECT ON PROGRAM	4-19
DEF	USER FUNCTION DEFINITION	4-22
DIM	DIMENSION ARRAY VARIABLES	4-20

ASSIGNMENT AND PROGRAM CONTROL		
LET	VARIABLE ASSIGNMENT	4-23
GOTO	EXECUTE FROM GIVEN PROGRAM LINE	4-24
IF	CONDITIONAL PROGRAM EXECUTION	4-25
ELSE	CONTINUES FROM 'IF-THEN'	4-26
GOSUB	CALL A BASIC SUBROUTINE	4-27
POP	REMOVE LAST 'GOSUB' RETURN ADDRESS	4-27
RETURN	EXIT FROM A BASIC SUBROUTINE	4-27
ON	MULTI-WAY CONDITIONAL GOTO	4-33
FOR	EXECUTE A LOOP	4-34
NEXT	INDICATES THE END OF A 'FOR' LOOP	4-34
ERROR	TRAP ERRORS TO SUBROUTINE	4-40
ESCAPE	ENABLE OPERATION OF ESCAPE KEY	4-72
NOESC	DISABLES OPERATION OF ESCAPE KEY	4-72
STOP	STOP THE PROGRAM	4-41
END	END OF PROGRAM	4-41

APPENDIX C

BASIC error messages

01 - Syntax Error

The line entered is not a correct BASIC statement.

02 - Unmatched delimiter

Some delimiters, such as quotes, go in pairs at the beginning and end of data. This error means only one was found.

03 - Invalid line number

BASIC was expecting a line number. You either typed something completely different or the number was too large.

04 - Illegal variable name

Variable names may have up to three letters or one letter and a number from 1 to 127.

05 - Too many variables

There may only be 140 different variable names in one BASIC program. Try moving a number of separate variables into one array. If you get this message when you have only used a few variables then your program has become corrupted.

06 - Illegal character

Some characters are not used by BASIC, such as '|'. They may only be used in quotes as part of a string.

07 - Expecting operator

The next thing to be typed in should have been an operator, such as '='. This error happens, for example, when you type a fourth letter in a variable name.

08 - Illegal function name

User defined functions may only be called FNA to FNZ.

09 - Illegal function argument

The argument to a user defined function may only be numeric.

10 - Out of memory

Your program has used all available memory. Check that your program has not corrupted itself. Try reducing array dimensions.

11 - Stack overflow

Either FOR's or GOSUB's are nested too deeply. Rewrite that part of the program to keep within nesting limitations.

12 - Stack underflow

A RETURN has been executed without a GOSUB to return to.

13 - No such line number

A transfer statement, such as GOTO or GOSUB, uses a line number that doesn't exist. Check the statement and the one it should jump to.

14 - Expecting string variable

A string variable should be entered here.

15 - Invalid screen command

An invalid screen command has been attempted, such as using a non existent shape table entry or an out of range sprite number.

16 - Expecting dimensioned variable

An array should be used here.

17 - Subscript out of range

An attempt has been made to access an array element with a subscript greater than the maximum declared in the DIM statement.

18 - Too few subscripts

The number of subscripts in a reference to an array must match the number in the DIM statement.

19 - Too many subscripts

The number of subscripts in a reference to an array must match the number in the DIM statement.

20 - Expecting simple variable

An array may not be used here.

21 - Digits out of range

Digits not in expected range.

22 - Expecting variable

A variable must be used here.

23 - Read out of Data

There are no more DATA statements left for a READ statement to read from.

24 - Read Data types differ

The value in the DATA statement must be of the same type (numeric or string) as the variable into which it is to be read.

25 - Square root of negative number

You cannot take the square root of a negative number.

26 - Log of non-positive number

You cannot take the log of a non positive number.

27 - Expression too complex

BASIC cannot handle an expression this complicated. Split it down into several calculations.

28 - Division by zero

Division by zero would give an infinite result.

29 - Floating point overflow

The maximum floating point number has been exceeded.

30 - Range error

Integers may only have a value between -32768 and +32767.

31 - Missing NEXT

A FOR statement must have a NEXT to mark the end of the loop. Note that NEXT should be the first statement on a line.

32 - Missing FOR

You cannot end a loop until you have begun it. Each NEXT must have a FOR somewhere before it to start the loop going. FOR-NEXT loops must not 'cross over' each other.

33 - EXP has invalid argument

Check that this is really the number you wish to raise e to the power of.

34 - Corrupted number

The value in this variable appears corrupted. This can happen, for example, when you take a variable containing a string and try and use it as a number.

35 - Parameter error

One or more of the parameters is incorrect.

36 - Missing assignment operator

An '=' sign is needed here. BASIC did not find one where it was expected.

37 - Illegal delimiter

The wrong type of symbol has been used to delimit items.

38 - Undefined function

A REF statement must be executed before a user defined function may be called.

39 - Undimensioned variable

A DIM statement must be executed to dimension an array before that array can be used.

40 - Undefined variable

A variable must have a value before it can be used.

42 - Interrupt without trap

An interrupt has been received for which there is no trap. This can happen with a noisy mains supply.

43 - Invalid baud rate

Only baud rates between 75 and 19200 are valid for a 9902.

44 - Illegal opcode

The Cortex has attempted to execute an undefined instruction. This normally only happens when you are working in machine code.

45 - EPROM verify error

The Cortex EPROMs have become corrupted.

46 - Invalid device number

The device specified in the UNIT statement does not exist on this Cortex.

47 - Required hardware not found

Extended commands require the memory mapper chip and E-bus interface to be populated.

48 - Illegal in current mode

Some commands may not be executed from within a program.

49 - Invalid address

Some commands, such as NEW, can be given an address. Only certain addresses are valid. See the section for the command in use.

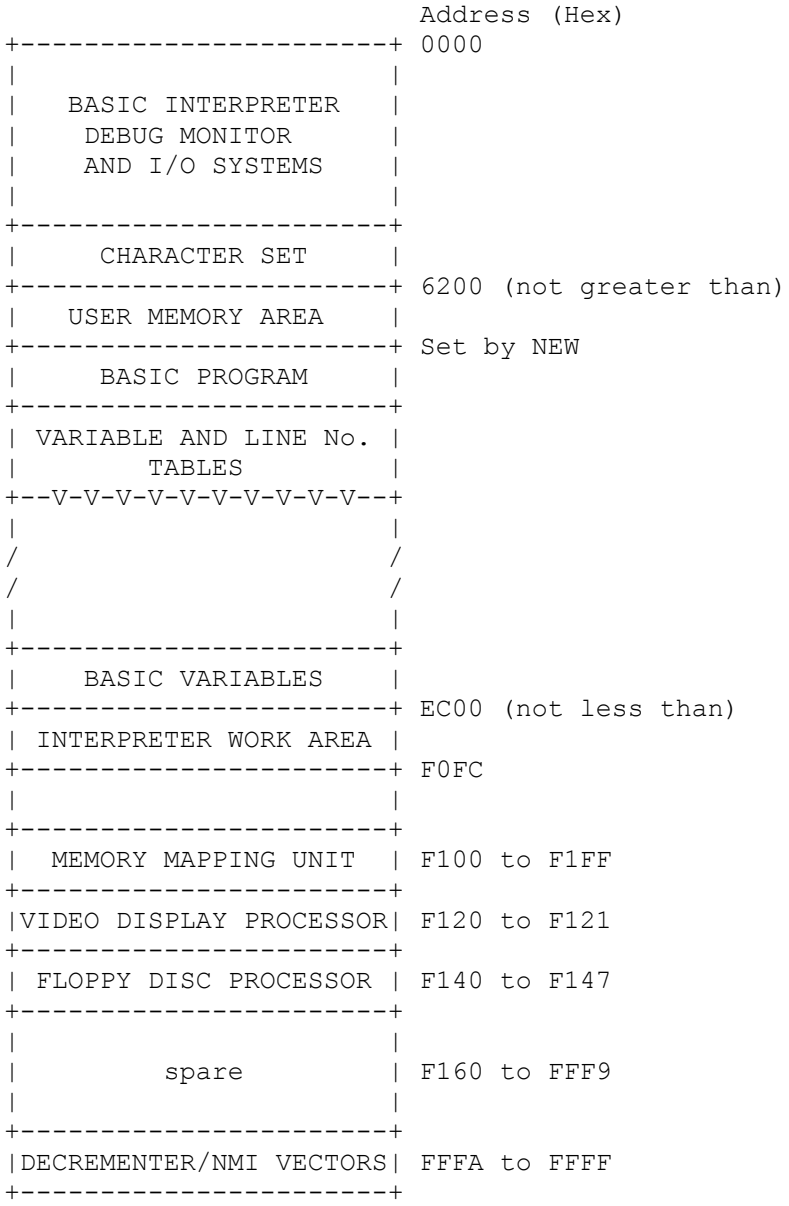
APPENDIX D

List of monitor commands

APPENDIX E

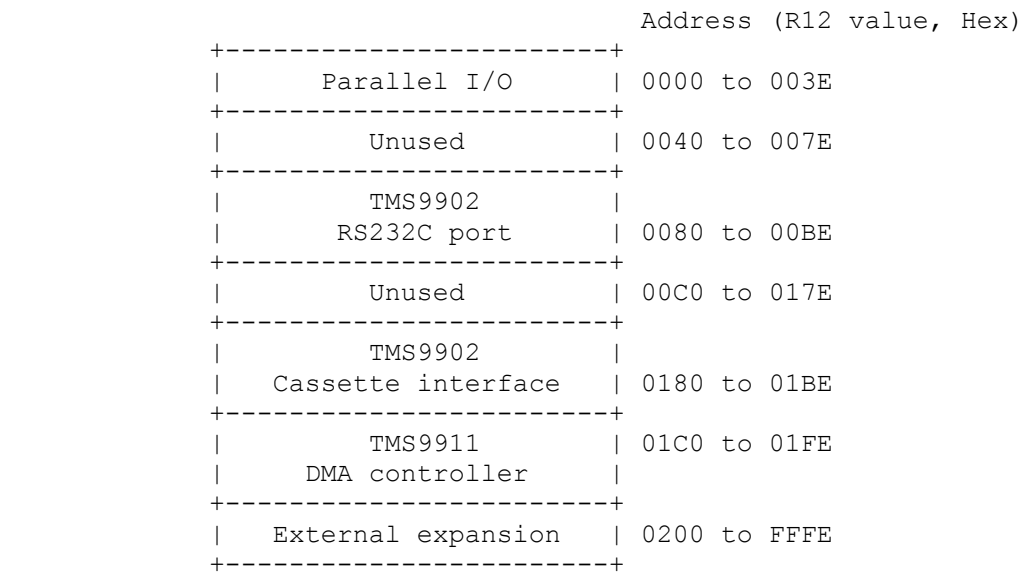
Memory and CRU maps

System memory map



Areas not used other than that marked spare will decode to multiple definitions of existing devices.

CRU map



The parallel I/O from 0000 to 003E breaks down as follows:

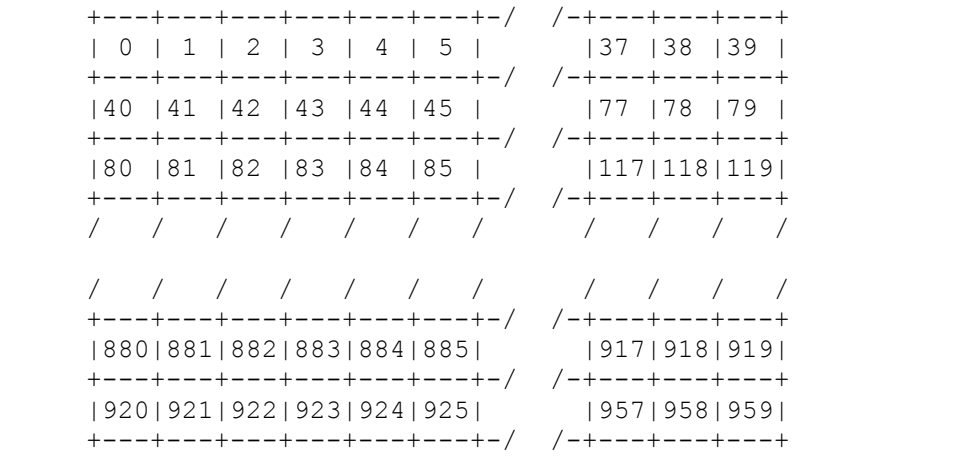
	OUTPUT	INPUT
BIT 0	CLOCK ACTIVITY	UNUSED
BIT 1	KBD INTERRUPT RESET-	UNUSED
BIT 2	BUS INTERRUPT RESET-	DISC SIZE
BIT 3	BUS TIMEOUT ENABLE	DISC DENSITY
BIT 4	DRIVE SIZE	FDC INTERRUPT-
BIT 5	ROM 0N-	KBD INTERRUPT-
BIT 6	BELL	VDP INTERRUPT-
BIT 7	UNUSED	BUS INTERRUPT-
BIT 8-15	MULTIPLY DEFINED	KBD DATA
BIT 16-31	MULTIPLY DEFINED	MULTIPLY DEFINED

Minus sign after signal indicates active low

APPENDIX F

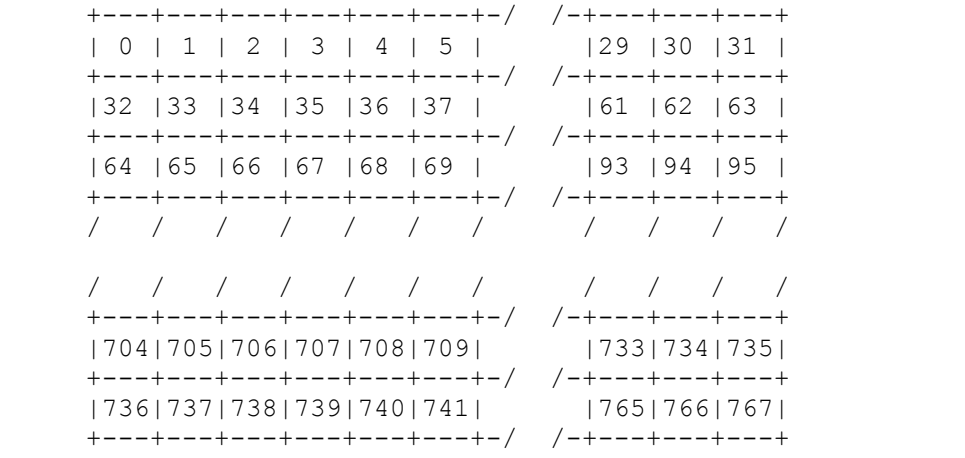
Screen layout and coordinates

Character cells in text mode:



CHARACTER CELL No.= HORIZONTAL POSITION + 40*VERTICAL POSITION
In TEXT mode there are 24 rows (0 TO 23) of 40 chars (0 TO 39)

Character cells in graphic mode:



CHARACTER CELL No.= HORIZONTAL POSITION + 32*VERTICAL POSITION
In GRAPH mode there are 24 rows (0 T0 23) of 32 chars (0 TO 31

Pixel coordinates:

In graphics mode there are 256 pixels horizontally (0 TO 255) and 192 pixels vertically. (0,0) is top left corner of the screen.

APPENDIX G

TMS9995 instruction set

Symbols Used

G,G1,G2 - General memory addresses
R – Workspace register address
S – Symbolic memory address
E - Expression (all symbols previously defined)
I – Immediate value
T - Term (range 0 - 15)
( ) - Contents of the address within parenthesis
-> - 'Replaces'
: - 'Is compared to'
C – Count (0 – 15)
* - Result is compared to zero

Additional symbols for 9989, 9995 and 99000

R* - Register pair R1 and R2
G1,G1+2 - General memory address double word

Instruction	Opcode	Format Type	Status Bits Affected	Format		Effect
ABSOLUTE VALUE	0740	6	*0-2,4	ABS	G	ABSOLUTE(G)->(G)
ADD BYTES	B000	1	*0 -- 5	AB	G1,G2	(G1)+(G2)->(G2)
ADD IMMEDIATE	0220	8	*0 -- 4	AI	R,I	(R)+I->(R)
ADD WORDS	A000	1	*0 -- 4	A	G1,G2	(G1)+(G2)->(G2)
AND IMMEDIATE	0240	8	*0 -- 2	ANDI	R,I	(R)AND I->(R)
BRANCH	0440	6		B	G	G->(PC)
BRANCH AND LINK	0680	6		BL	G	G->(PC) (PC)->(R11)
BRANCH AND LOAD WP	0400	6		BLWP	G	(G)->(WP) (G+2)->(PC) (Old WP)->(R13) (Old PC)->(R14) (Old ST)->(R15)
CLEAR	04C0	6		CLR	G	0->(G)
CLOCK OFF	03C0	7		CKOF		External
CLOCK ON	03A0	7		CKON		External
COMPARE BYTES	9000	1	0-2,5	CB	G1,G2	(G1):(G2)
COMPARE IMMEDIATE	0280	8	0 -- 2	CI	R,I	(R):I
COMPARE WORDS	8000	1	0 -- 2	C	G1,G2	(G1):(G2)
COMPARE ONES CORRES.	2000	3	2	COC	G,R	ST2=AND of RBITS corres. to GBITS=1
COMPARE ZEROS CORRES.	2400	3	2	CZC	G,R	ST2=NAND of RBITS corres. to GBITS=1
DECREMMNT BY ONE	0600	6	*0 -- 4	DEC	G	(G)-1->(G)
DECREMENT BY TWO	0640	6	*0 -- 4	DECT	G	(G)-2->(G)

Instruction	Opcode	Format Type	Status Bits Affected	Format	Effect
DIVIDE	3C00	9	4	DIV G,R	INT(R)/(G)->(R) REM(R)/(G)->(R+1)
EXECUTE INSTRUCTION	0480	6		X G	Execute instr at G
EXTENDED OPERATION	2C00	9	6	XOP G,T	(>40+4*T)->(WP) (>42+4*T)->(PC) Eff add of G->(R11) (Old WP)->(R13) (Old PC)->(R14) (Old ST)->(R15) 1->ST6
EXCLUSIVE OR	2800	3	*0 -- 2	XOR G,R	(G) XOR (R)->(R)
IDLE	0340	7		IDLE	IDLE; External
INCREMENT BY ONE	0580	6	*0 -- 4	INC G	(G)+1->(G)
INCREMENT BY TWO	05C0	6	*0 -- 4	INCT G	(G)+2->(G)
INVERT BITS	0540	6	*0 -- 2	INV G	1s COMP(G)->(G)
JUMP (UNCONDITIONAL)	1000	2		JMP S	S->(PC)
JUMP IF CARRY	1800	2		JOC S	S->(PC) IF ST3=1
JUMP IF EQUAL	1300	2		JEQ S	S->(PC) IF ST2=1
JUMP IF GREATER THAN	1500	2		JGT S	S->(PC) IF ST1=1
JUMP IF HIGH OR EQUAL	1400	2		JHE S	S->(PC) IF ST0=1 OR ST2=1
JUMP IF LESS THAN	1100	2		JLT S	S->(PC) IF ST1=0 AND ST2=0
JUMP IF LOGICAL HIGH	1B00	2		JH S	S->(PC) IF ST0=1 AND ST2=0
JUMP IF LOGICAL LOW	1A00	2		JL S	S->(PC) IF ST0=0 AND ST2=0
JUMP IF LOW OR EQUAL	1200	2		JLE S	S->(PC) IF ST0=0 OR ST2=1
JUMP IF NO CARRY	1700	2		JNC S	S->(PC) IF ST3=0
JUMP IF NO OVERFLOW	1900	2		JNO S	S->(PC) IF ST4=0
JUMP IF NOT EQUAL	1600	2		JNE S	S->(PC) IF ST2=0
JUMP IF ODD PARITY	1C00	2		JOP S	S->(PC) IF ST5=1
LOAD CRU	3000	4	*0-2,5	LDCR G,T	T bits (G) -> CRU
LOAD IMMEDIATE	0200	8	*0 -- 2	LI R,I	I->(R)
LOAD INTERRUPT MASK	0300	8	12-15	LIMI I	I->(Int. mask)
LOAD ROM AND EXECUTE	03E0	7	12-15	LREX	External
MOVE BYTE	D000	1	*0-2,5	MOVB G1,G2	(G1)->(G2)
MOVE WORD	C000	1	*0 -- 2	MOV G1,G2	(G1)->(G2)
MULTIPLY	3800	9		MPY G,R	MSW((G)*(R))->(R) LSW((G)*(R))->(R+1)
NEGATE	0500	6	*0 -- 4	NEG G	-(G)->(G)
OR IMMEDIATE	0260	8	*0 -- 2	ORI R,I	(R) OR I ->(R)
RESET I/O	0360	7		RSET	External
RETURN WORKSPACE POINTER	0380	7	0 -- 6 12-15	RTWP	(R13)->(WP) (R14)->(PC) (R15)->(ST)

Instruction	Opcode	Format Type	Status Bits Affected	Format	Effect
SET BIT TO ONE	1D00	2		SBO E	1->(E+(R12))
SET BIT TO ZERO	1E00	2		SBZ E	0->(E+(R12))
SET TO ONES	0700	6		SETO G	>FFFF->G
SET ONES CORRES. BYTE	F000	1	*0-2,5	SOCB G1,G2	(G1) OR (G2) ->(G2)
SET ONES CORRES. WORD	E000	1	*0 -- 2	SOC G1,G2	(G1) OR (G2) ->(G2)
SHIFT LEFT ARITH. ◊	0A00	5	0 -- 4	SLA R,C	Shift left C bits and '0' fill
SHIFT RIGHT ARITH. ◊	0800	5	0 -- 3	SRA R,C	Shift right C bits and MSB fill
SHIFT RIGHT CIRCULAR ◊	0B00	5	0 -- 3	SRC R,C	Shift right C bits and LSB into MSB
SHIFT RIGHT LOGICAL ◊	0900	5	0 -- 3	SRL R,C	Shift right C bits and '0' fill
STORE CRU	3400	4	*0-2,5	STCR G,T	T CRU bits ->(G)
STORE STATUS REGISTER	02C0	8		STST R	(ST)->(R)
STORE WORKSPACE POINTER	02A0	8		STWP R	(WP)->(R)
SUBTRACT BYTE	7000	1	*0 -- 5	SB G1,G2	(G2)-(G1)->(G2)
SUBTRACT WORD	6000	1	*0 -- 4	S G1,G2	(G2)-(G1)->(G2)
SWAP BYTES	06C0	6		SWPB G	Interchange bits 0-7 With bits 8-15 of G
SET ZEROES CORRESPONDING BYTE	5000	1	*0-2,5	SZCB G1,G2	(INV(G1)) AND (G2) ->(G2)
SET ZEROES CORRESPONDING WORD	4000	1	*0 -- 2	SZC G1,G2	(INV(G1)) AND (G2) ->(G2)
TEST BIT	1F00	2	*0 -- 4	TB E	(R12)+E->ST2

◊ If C=0 then count taken from bits 12 - 15 of R0.
If this is zero then C=16.

Instruction	Opcode	Format Type	Status Bits Affected	Format	Effect
LOAD ST FROM REGISTER	0080	8	0 - 15	LST R	R->ST
LOAD WP FROM REGISTER	0090	8		LWP R	R->WP
SIGNED DIVIDE	0180	6	*0-2,4	DIVS G	INT(R*)/(G)->(R0) REM(R*)/(G)->(R1)
SIGNED MULTIPLY	01C0	6	*0 -- 2	MPYS G	MSW((R*)*(G))->(R0) LSW((R*)*(G))->(R1)

APPENDIX H

ASCII codes

Char	Hex	Char	Hex	Char	Hex
NUL	00	+	2B	V	56
SOH	01	,	2C	W	57
STX	02	-	2D	X	58
ETX	03	.	2E	Y	59
EOT	04	/	2F	Z	5A
ENQ	05	0	30	[	5B
ACK	06	1	31	\	5C
BEL	07	2	32	]	5D
BS	08	3	33	^	5E
HT	09	4	34	_	5F
LF	0A	5	35	`	60
VT	0B	6	36	a	61
FF	0C	7	37	b	62
CR	0D	8	38	c	63
SO	0E	9	39	d	64
SI	0F	:	3A	e	65
DLE	10	;	3B	f	66
DC1	11	<	3C	g	67
DC2	12	=	3D	h	68
DC3	13	>	3E	i	69
DC4	14	?	3F	j	6A
NAK	15	@	40	k	6B
SYN	16	A	41	l	6C
ETB	17	B	42	m	6D
CAN	18	C	43	n	6E
EM	19	D	44	o	6F
SUB	1A	E	45	p	70
ESC	1B	F	46	q	71
FS	1C	G	47	r	72
GS	1D	H	48	s	73
RS	1E	I	49	t	74
US	1F	J	4A	u	75
Space	20	K	4B	v	76
!	21	L	4C	w	77
"	22	M	4D	x	78
#	23	N	4E	y	79
$	24	O	4F	z	7A
%	25	P	50	{	7B
&	26	Q	51	|	7C
'	27	R	52	}	7D
(	28	S	53	~	7E
)	29	T	54	DEL	7F
*	2A	U	55		

APPENDIX I

Bibliography

The following publications by Texas Instruments are referred to in this manual. They are not required to use the CORTEX but are very useful for advanced software development and designing add on hardware.

PUBLICATION	AUTHOR
TMS9900 family data book	Texas Instruments
Software development handbook	Geoff Vincent & Jim Gill
E—bus system design handbook	H. Althoff, H. Hirsch et al.