CHAPTER 1

NOTE: The Commodore 700 refers to the B-128.

INTRODUCTION

The Commodore 700 computers are among the most modern microcomputers in the world. Commodore has an international reputation for technological innovation and this can be seen in the exceptional design and price/performance ratio offered by these computers.

These computers - the CBM 700s - represent the further development of existing models, including improvements in hardware and software which are totally original. These are some of the most important features of the CBM 700:-

- User memory size 128K
- Enlarged BASIC 4.0+ interpreter
- Screen with 25 lines each with 80 columns (program lines are not limited to 80 characters)
- Fully programmable three voice sound synthesizer
- Serial interface RS232

If the computer is to be used in the office or in professional surroundings, you will profit from the advantages of the new and extended BASIC 4.0+. This extension includes automatic processing of the greatly increased memory, a highly developed "error tracker" as well as the implementation of the PRINT USING command and the IF...THEN...ELSE program structure.

The CBM 700 screen with its 25 lines, each with 80 columns has the standard format for efficient, professional program packages in areas such as word processing, accounting, information processing, data transfer, auditing and finance.

1.1 CBM 700 Enlarged Memory

One of the most important features of the Commodore 700 computer is the memory which forms the heart of the 700 range thanks to the progressive technology of the 6509 microprocessor. The 6509 has 20 address lines, compared to the 16 lines of other, less efficient, microprocessors. The four extra address lines mean that the 700 can address sixteen times as much memory.

Some 700 machines have 128K of memory fitted as standard; others have 256K memory as standard.

The banks (0 and 15) are reserved for the 6509 and the 700 operating system.

1.2 BASIC 4.0 plus.

The Commodore 700 computers are equipped with a considerably enlarged BASIC 4.0+ interpreter. BASIC is the most widely used programming language for microcomputers. There are thus thousands of BASIC programs for almost every conceivable application.

However, programs designed and written by you are also possible with this language.

The enlarged and improved BASIC interpreter is built into every CBM 700 as ROM (Read Only Memory). Your new computer needs only to be switched on and a BASIC program can be started.

The programmer does not need to consider the memory processing. The BASIC interpreter will use the memory automatically. The increase in available memory permits BASIC programs which can cope with more work at increased speed.

Additional possibilities with the new BASIC 4.0+ interpreter are:-

- VDU commands
- Formatted data output.
- IF...THEN...ELSE program structures
- Editing and directory processing
- Variables and data processing
- RS232 interface
- Memory processing

1.3 Sound Effects and Music.

The Commodore 700 has one of the most modern digital sound chips: the 6581 sound interface device (SID). This contains:-

- 3 independent programmable sound generators
- 3 envelope generators
- programmable filter

Each of the three generators has its own programmable oscillator and wave generator. Each one also has its own envelope generator with which the amplitude of the signal (volume) can be defined as a function of time. It is thus possible to simulate simple characteristic waves for many musical instruments. Completely new sounds can also be produced. All three envelope generators are connected to a programmable filter - this can be programmed as high pass, band pass or low pass. This filter is probably the most important feature of the synthesizer since very complex sounds can be produced by simple programming. All tone generators can be interconnected for synchronisation or ring modulation effects to make the production of very interesting and unusual sounds simple.

1.4 Serial Interface.

The 700 has an RS232 interface. This enables the connection of many of the printers, terminals and modems on the market.

The 6551 asynchronous interface (ACIA) is responsible for the RS232 interface. The new BASIC 4.0+ interpreter has software to program this interface easily. A channel is simply opened and used, as for a file or a printer, with the standard Input/Output commands in BASIC.

1.5 Installation.

None of the models in the 700 range have any special requirements as regards temperature. The computers function in every climate even where you personally may find it only bearable.

The electrical side also presents no problem. The mains supply has enough capacity to "smooth" larger deviations in the current or voltage peaks. Disturbances may only happen whilst switching on if very large electric motors are close by.

Worries about electrical supply to your computer need also not concern you, since it takes only the same amount of current as two normal desk lights (about 130 Watts).

To be thorough, however, it must also be noted that very high radio activity or "hard x-rays" can lead to problems.

1.6 Setting Up.

Make sure your computer is switched off before beginning installation. Also check that the monitor is switched off in low profile models - pay attention to the operation instructions for the monitor supplied with the machine. The mains switch is situated at the back of the computer. Starting a computer in the low profile range varies from that in the high profile range only in the first point. Your B-128 System is low profile.

1. Low profile - Connect your computer to the monitor. Use a video cable. There is a 5-pin socket for this at the rear. High profile - Connect the keyboard to the socket on the front of the computer. The Commodore logo on the plug should be uppermost.

2. Next, your peripherals must be connected. For this you need an IEEE to Edge connector cable. The edge connector of this lead goes to the IEEE socket on the computer. The writing on this plug should be uppermost. The other plug is connected to one of the peripherals. For each further connection of a peripheral a further IEEE to IEEE cable is required. One of the ends is pushed onto the plug of a peripheral already connected (pick-a-back) and the other joined to the new peripheral. (Ensure that the securing screws are tightened so that the plugs sit squarely upon one another).

3. Now you can connect the mains electricity lead. Your computer is ready for operation.

1.7 On/Off Switch.

The on/off switch on the Commodore 700 computer is on the back of the machine.

When the computer is switched on, a test routine is run, during which the computer checks itself for errors. After 4-6 seconds (depending on memory type) the "ready" message will appear. Your machine is now ready to go and you can start straight away. (Should the computer fail to work, try again. If it still fails, consult your dealer).

Before switching off, ensure that you have saved your data (i.e. transferred it onto disk), if you need it at a later date. The same applies, of course, to programs you have written yourself.

CHAPTER 2.

THE KEYBOARD.

The 700 has a keyboard which is very similar to a typewriter keyboard. However, on a closer inspection you will discover a whole range of keys and characters not found on a normal typewriter.

2.1 RETURN and ENTER.

These keys enter data into the computer and/or start processing of data. They have the same effect.

2.2 SHIFT

This key corresponds to the SHIFT key on a normal typewriter. If you press the shift key at the same time as a letter key, you will obtain the corresponding letter in capitals or, with keys having 2 characters, the top character. Having switched your micro to the graphic mode however, capital letters will appear without pressing the SHIFT key and if SHIFT is pressed, the graphic characters on the front of the key will be obtained.

2.3 SHIFT LOCK.

This is a standard Shift Lock key.

2.4 OFF/RVS.

After pressing this key, all subsequent characters are displayed in inverse (REVERSE) video, therefore what is normally light becomes dark and vice-versa. When this key is pressed simultaneously with the shift key, the reverse mode is switched off again.

2.5 NORM/GRAPH.

This key selects the graphics character set of your computer. Instead of small and capital letters, capitals and a set of graphic characters appear. The special characters are shown on the front of the key and are reached by using SHIFT (letters) or by CTRL (other keys). The NORM/GRAPH together with SHIFT will switch the VDU screen back to capital/small letters (normal display).

2.6 Cursor Control Keys.

These keys move the cursor (which is the position where the next character will appear) in the direction shown on the key. Keep the key depressed to move the cursor over longer distances. i.e. repeat.

2.7 INST/DEL.

When this key is pressed the character immediately before the cursor is erased and all subsequent characters on that line are moved to the left to fill the gap.

"Line" here means the logical line - i.e. all the characters which have been entered into the computer until a RETURN or ENTER key is pressed. This "logical Line" could fill the entire VDU screen, but the computer can only interpret lines of up to 160 characters. The INST/DEL with the SHIFT key produces a free space at the position marked by the cursor. All characters following will be pushed one space to the right.

2.8 CLR/HOME.

This key moves the cursor back to its start position top left (HOME). If it is used with the shift key the entire screen is cleared (CLEAR). By depressing the CLR/HOME key twice, any predefined window is cleared and control of the whole screen is re-established.

2.9 CTRL

When pressed during a scroll in direct mode this key slows the scroll rate. When pressed together with other non-letter keys, the character on the front of the key is displayed. A range of special functions is possible with some letter keys.

Functions of the control key:-

	Without SHIFT	with SHIFT
d	Delete	Delete
g	Enable bell	-
i	Tab	Tab
m	Return	Return
n	Set text mode	-
o	Set top	Set top
q	Cursor down	Cursor down
r	RVS on	RVS on
s	HOME	HOME
t	Delete	Delete

2.10 RUN/STOP

This key interrupts a program (if the programmer has not entered this function into the program already). Pressed together with SHIFT (in direct mode), the first program from disk in drive 0 is loaded and started.

2.11 Commodore Key

When listing programs or data output, the screen display is automatically "rolled" upwards (SCROLL) when the lowest line is reached. By pressing this key, the scrolling is stopped, and is started again by any other key.

2.12 ESC Key.

This key resets from quotes mode and insert mode.

The computer is in quotes mode after pressing the " key (double inverted commas). After this, some of the special keys mentioned above no longer function as described, but the screen shows that the relevant key has been pressed. This mode is ended by pressing the " again or by using the ESC key. The purpose of this mode is to store the control keys in a string for later display. (See PRINT).

If SHIFT-INST/DEL are used together, the computer is switched to insert mode. Here also, the cursor movements are not displayed directly, but the key-pressing action is stored (the insert could have occurred in a string in inverted commas).

The insert mode is switched off when all available places are filled or by using the ESC key. The ESC key has a further special function. When it is pressed the following letters generate functions of their own:-

Letter Key	Special Function
a	Sets Insert mode on.
b	Sets bottom RH corner of the text window at cursor position.
c	Resets insert mode off.
d	Deletes cursor line and closes up from below.
e	Non-flashing cursor selected.
f	Resets normal cursor (flashes).
g	Sets internal bell on (enable).
h	Resets internal bell off (disable).
i	Inserts a line on cursor line and moves text down.
j	Moves cursor to start (LH) of text on line.
k	Moves cursor to end (RH) of text on line.
l	Reset wrap mode off (enable scroll).
m	Set wrap mode on (disable scroll).
n	Reset screen to normal video.
o	Clear quotes and RVS, but not insert mode.
p	Erase to start (LH) of cursor line.
q	Erase to end (RH) of cursor line.
r	Set screen to reverse video.
s	Reset solid cursor (from underscore).
t	Sets top (LH) corner of text window.
u	Sets underscore cursor.
v	Scroll vertically up one line.
w	Scroll down one line.
x	Reset from ESC sequence (as if you had never pressed ESC).
y	Select normal character set chip.
z	Select alternate character set chip.

Note: y and z only have an apparent effect if the character sets are not identical.

2.13 Numeric Keypad.

Sometimes you will want to use your 700 simply as a calculator. All keys for this purpose are situated together on the RH side of the keyboard (some are repeated on the main keyboard and have the same effect).

On the keypad, with the exception of the ENTER key, all the keys have the same function with or without SHIFT. Apart from the ten numbers, you will find a decimal point, a double zero (for convenience), the four calculation signs +, -, *, / and the CLEAR ENTRY (CE) key with which you can erase the last number typed. Do not use commas or colons in numbers.

The Question Mark key may be used as an abbreviation for the word PRINT.

2.14 Function Keys.

Finally, there are 10 further useful keys - F1 to F10 - which are situated top left on your keyboard. Each one of these keys can take a command, a text or even a whole program, according to your requirements. Each key may be used twice, since, when used with the SHIFT key, each one of these function keys receives a second meaning (F11 to F20). The functions allocated to each key are listed on the screen after the command KEY. After switching on, type KEY and then RETURN and the following list will appear:-

```
Key 1,"print"
Key 2,"list"
Key 3,"dload"+chr$(34)
Key 4,"dsave"+chr$(34)
Key 5,"dopen"
Key 6,"dclose"
Key 7,"copy"
Key 8,"directory"
Key 9,"scratch"
Key 10,"chr$("
```

Keys 11-20 (attainable together with SHIFT) are not defined at power on. You can change the list at any time and also define keys 11-20. For example, if you want to use F11 so that a BASIC program from line 300 will be LISTed. You must obtain a free line on the screen, type KEY 11, "LIST-300" + CHR$ (13). Now press SHIFT F1 (F11) and the program will list starting at line 300. Conclude your entry with RETURN.

Function keys remain programmed until the machine is turned off.

CHAPTER 3

INTRODUCTION TO THE NEW ENLARGED BASIC 4.0+

The 700 series computers are equipped with a considerably enlarged BASIC 4.0+ interpreter. The new BASIC 4.0+ permits problems to be solved by using individual programs exactly tailored for the purpose. Whatever the solution, the new BASIC 4.0+ with a built-in screen editor will do it quickly, easily and without problems. The interpreter is built into every 700 computer as ROM(Read Only Memory). This means that when the machine is switched on BASIC programs can be loaded and started immediately.

This enormous memory capacity means that BASIC programs can deal with more work more efficiently. Complicated algorithms for data exchange between working storage and mass storage are no longer necessary, as there is enough available work space. Room for comprehensive error trapping in the user program no longer poses a problem. It is now possible to use programs which previously were only associated with very large machines. The most important features of the new interpreter are:-

- Screen commands
- Formatted data output
- IF..THEN..ELSE structures
- Editing and directory processing
- Variable and data processing
- Error trapping
- Memory processing

3.1 Formatted data output

Processing programs need the facility to easily format print-outs and tables. Commodore has therefore implemented the PRINT USING statement. The number format on the printer or in a file is easily defined with this statement. The most important features are:-

- Positioning of numeric sign
- Positioning of commas and decimal points
- Exponent output
- Positioning of text

3.2 IF..THEN..ELSE Structures

The IF..THEN..ELSE structure is a very useful element in every programming language. Existing programs which contain these structures may now be used with the new interpreter. To accentuate its efficiency, we will take a simple example:-

Variable C should be assigned the value of variables A or B, depending on the larger of the two. Without the IF..THEN..ELSE statement, the solution for this simple problem would be:-

```
IF A>B THEN C = A
IF B>A THEN C = B
```

C=B: IF A>B THEN C=A

Using the IF..THEN..ELSE statement, however, the solution is simplified:-

```
IF A>B THEN C = A: ELSE C = B
```

This simplification makes the program quicker, easier to understand and simplifies the changing or expansion of an existing program. This in turn saves time and money.

3.3 Editing Function, Directory Processing

The new BASIC has a DELETE command in order to erase BASIC program lines. For example:-

```
DELETE 10-100
```

can be entered to erase all program lines between 10 and 100.

The new DIRECTORY command presents a list of all files in the disk. For example:-

```
DIRECTORY "edu*"
```

(The * is a pattern matching symbol-see the disk drive manual). This command will only fetch those file names beginning with the letters "edu".

3.4 Variable and Data Processing

The interpreter also offers an enhanced RESTORE statement in conjunction with DATA and READ statements. Sometimes it is necessary to re-read certain parts of DATA statements. With the new RESTORE, the line number of the DATA statement to be read by the next READ operation can be given. For example:-

```
RESTORE 5000
```

Sets the DATA pointer to the first item in the DATA statement in line 5000. Additionally, the interpreter has the string function INSTR. Using this, one string can be sought within another example:-

```
10 A$= "FIND THIS STRING"
20 LOC = INSTR (A$,"THIS")
```

The variable LOC now receives value 6 - the start position of the word "THIS" in A$.

3.5 Error Trapping.

Sometimes it is sensible to trap errors which are normally processed by BASIC, for example division by zero. In this instance BASIC would normally give an error message and stop the program. If a TRAP statement is used, such an error can be dealt with by the program itself, allowing you to restart the program where the error occurred. There are several ways of treating an error. Variables can be corrected in the statement and re-executed. The program execution can also be restarted at another point. Error trapping in BASIC 4.0+ also gives information on the type of error, on the line number in which it occurred and, if necessary, the text of the standard BASIC error message which BASIC would have displayed if the error had not been trapped.

3.6 RS232 Interface.

The 700 is equipped with an RS232 interface as standard. This interface allows connection with numerous types of printers, screens and modems. The transfer procedure is internationally standardised. Using the interface in BASIC is very simple: after opening a data channel for the interface with an OPEN statement, further programming takes place with PRINT or INPUT statements, as used for a printer or disk.

3.7 Memory Processing.

In order to make use of all of the memory, some commands from the BASIC interpreter have been enlarged and others added. These commands and statements permit:-

- Direct working with PEEK and POKE statements in specified areas of the enlarged memory,

- BLOAD or BSAVE commands for specified areas of the enlarged memory,

- Detection of the free memory space in certain areas of the enlarged memory.

CHAPTER 4

DATA TYPES IN BASIC.

Programs in every processing language process data. The interpreter in the enlarged BASIC 4.0+ uses three data types: real, integer and string. Arrays can be defined of each of these types. An array is a combination of elements of the same type in a form which can be visualised as a table of data. Generally, real numbers are used to present fractional numbers - i.e. numbers which have places after the decimal point as in 100.8899 or -0.66. Integer expressions have no places after the point, as in 10 or -3. Strings are used to present letters or text, for example:-

"Fred Bloggs" or "This is text".

4.1 Variables in BASIC.

Each variable receives its own name. A variable name consists of up to 159 alpha-numerical characters and must start with a letter. The last character may be a special character to determine the type of variable. A variable name may not contain BASIC commands, for example:- TOMATO is a syntax error because it contains the BASIC word TO. Only the first two characters and, if present, the last special character are stored. Differing variables, therefore use this last character for identification. The data type is determined by the variable name. Real numbers are defined by the first two letters of the name for example:-

A1,BD,TD,I,J,K,Z8.

Integers are defined by the first two letters of the variable name and a % (percent) sign. for example:-

A1%,ZZ%,F8%,J%,INCREMENT%.

Strings are defined by the first two characters of the name and by a dollar sign ($) as last character for example:-

A1$,B$,AXE$.

The enlarged interpreter has several internally defined names and words. These reserved words must not be used as variable names. The reserved words are:-

- All function names
- Input/Output status (ST)
- Disk status (DS and DS$)
- Error status variables (EL and ER)
- Time variable (TI$)

NOTE: TI is not a reserved word.

4.2 Real Numbers.

The interpreter executes arithmetical operations in real format, even if integer expressions are included. In this way, all constants are stored in real format. A real number can be either a whole number or a number with decimal places, and can be positive or negative. For example:-

2.4442, -0.5555, 6.7893, 21, 778012, 441777.

Numerical data in this format have 5 bytes and are stored in two parts as mantissa and exponent. The mantissa and the exponent give the location of the decimal point. The Interpreter permits a resolution of more than 10 decimal places for the mantissa.

The exponential form is a compact format for very large or very small numbers. There are limits, however, for the absolute value of numbers in real form. These are:-

Largest absolute value: approx. 1.7E+38
Smallest absolute value: approx. 2.9E-39

If the maximum value is exceeded, the error message ? OVERFLOW appears. If the minimum value is undercut, the value of the variable becomes 0. An underflow error message does not appear. These limits are also applicable for internal intermediate results in arithmetic expressions. Exceeding the range in the intermediate results can be the reason for unexpected error messages.

4.3 Integers.

A further way of storing numerical data is to use the integer format. Integer variables are defined by a percent (%) sign as the last character of the variable name. Only integers may be stored in this format, with a positive or negative sign. For example:-

1, 4711, 32000, 8032, -5774, -22, 100.

As with real numbers, there are also limits for the absolute values of integers:-

Largest integer = +32767
Smallest integer = -32768

If this range is exceeded, the error message ? ILLEGAL QUANTITY will appear. All internal calculations use the real number format. Integer values are converted into real format before being used in a calculation. The result also appears in real format. If such a result is changed into an integer, the places after the decimal point are simply cut off and not rounded up or down. So the expression A% = 5.9/2 will round up value 2 for the integer variable A%, and will not round up the value to 3.

4.4 Character Processing.

The third data format is text format (string). It is defined by a dollar sign ($) as the last character of the variable name. Text variables have a string of text characters, one byte per character. The whole string of characters is referred to as a single variable. Text constants are put within inverted commas in order to be used in a BASIC program. For example:-

"Do you wish to continue?"
"123456789"
"BASIC 4.0+"
"Any number or a word"

A text variable may contain:-

- Alphabetical characters (A...Z, a...z)
- Numerical characters (0...9)
- Special characters ($/%:+-...)

The characters in a text variable are presented normally. The control characters are presented in a reverse video if they appear in a text variable. Text which is entered via the keyboard has a maximum length of 157 characters for each text variable. In addition, longer text variables can be produced by linking the contents of more than one text variable by concatenation (+ operator).

For example: "TEXT 1" + "TEXT 2" is "TEXT 1TEXT 2"

But there are limits here too, the maximum length of a text variable is 255 characters. If this length is exceeded, the error message ?STRING TOO LONG appears. BASIC 4.0+ has a whole series of functions to process text variables. There are functions to establish the length of a text variable, to scan for a certain text within a variable, to convert a text variable containing numerical characters into number format, and many others. A text variable must never be used in a numerical expression, even if it only contains figures. It must first be converted to a numeric format.

4.5 Arrays.

An array is a collection of elements of the same data type, as in a table. The whole array is described by a single name. Each element has a fixed position within the array and the position is determined by an index. Let us take as an example a class of no more than 50 students whose names are to be used in a program. It would be highly impractical to process 50 different variable names, one for each student. Instead, an array of 50 text variables is used and the processing becomes very simple. The DIM statement is used to define such an array in order to reserve the relevant memory space:-

```
DIM NAME$ (49)
```

The NAME$ array is uni-dimensional, and can be described using a single index. The index lies within the range 0-49. Larger or smaller values lead to an error message. Now the program may print the names of some students. This could look like this:-

```
PRINT NAME$(0)          to print the first name
PRINT NAME$(4)          to print the fourth name
PRINT NAME$(49)         to print the last name.
```

As you can see, a certain array element can be reached by entering the index number. In the example the indices were numerical constants but variables can also be used. To express the whole array a FOR..NEXT loop can be used:-

```
10 FOR I = 0 TO 49
20 PRINT NAME$(I)
30 NEXT I
```

This example shows the simplest form of a data array - unidimensional. The BASIC interpreter in the 700 allows for multi-dimensional arrays within the following limits:-

Maximum number of dimensions = 255
Maximum number of elements per dimension = 32767

Theoretically, therefore, one array could be dimensioned with 255 different indices of which each can assume the value 0-32767. If the maximum value defined for the index is exceeded, or if one tries to define a negative index, the error message ?BAD SUBSCRIPT appears. If one tries to define an array with more than 32767 elements per dimension, the error message ?ILLEGAL QUANTITY appears. If the defined array size exceeds the memory space available in the system, the error message ?OUT OF MEMORY appears. In BASIC 4.0+ the number range for the index starts with 0 and ends with the maximum value defined in the DIM Statement. So an array with the definition A (5) has 6, not 5 elements - the indices can be between 0 and 5. Unidimensional arrays with not more than 11 elements do not need to be previously defined by a DIM statement. The actual array size is limited by the available system memory.

In the 700, this size is some 64 Kbytes for a uni-dimensional array. To give an example of multi-dimensional arrays, let us expand the number of students' names to 10 different classes, each of which may have up to 50 students. The dimensioning of the array is now:-

```
DIM NAME$ (9,49)
```

In this dimensioning statement, the first index is used to address the class and the second to find one child within that class. One can imagine this as a table with 10 columns (0-9), one for each class and 50 lines (0-49), one for each student in the class. This array can take 500 students (10 columns * 50 lines). To find an individual student in this array, one could write:-

```
PRINT NAME$ (0,13) to find the 14th child in the first class
PRINT NAME$ (9,1) to find the 2nd child in the 10th class.
```

It is sometimes confusing to use the 0 element of an array. If no consideration of the memory limitation is to be taken, one can simply ignore this element and start the counting with 1, or use this element for special purposes (to form sums for example).

CHAPTER 5

STRUCTURE OF BASIC.

This chapter contains a summary of the fundamental elements of the programming language BASIC and, in particular, describes the language additions for the 7ØØ. If you are not already familiar with BASIC and would like to learn it, you should use one of the many introductions to BASIC which are readily available in bookshops (see bibliography). This chapter does not replace an introduction to the BASIC language. BASIC is an efficient and easily understood programming language,simplifying the creation of well-structured solutions to programming problems. Basic language statements are of several types:-

- Commands
- Statements/Expressions
- Functions

A command, an expression, or a function are given by specific keywords. The keyword is recognised by the BASIC interpreter during program processing and the operation associated with that keyword is executed. For example, in the statement PRINT A$, the keyword PRINT is recognised as a statement to print something. The section of the BASIC interpreter which is responsible for data print-out now analyses the rest of the statement (A$) in order to ascertain what should be printed. In this instance it is the contents of the string A$ which will appear on the screen.

The classification of BASIC keywords into commands, statements or functions depends on the type of action required by the interpreter. Commands are used in order to do something with the program. A program can be changed, listed, loaded, erased, started, etc. by a command. Statements are the words which make up the program. The computer is told by statements what it is to do during the program run. Functions perform operations that evaluate data for the program to process further. For example the length of a string can be determined by a function. Functions are always carried out as part of a Statement.

There are two ways to execute commands and statements in BASIC. Either they are executed as part of a BASIC program (program mode) or they are executed immediately after entry by entering them without line number (direct mode). A BASIC Program line always begins with a line number within the range Ø-63999. The entry of a statement without line number in a direct mode is very useful when looking for an error, one can see the value of the variables straight away and change them if necessary.

BASIC statements can be divided into four types:-

1. Declaration statements to define data and the user's own functions in the program.

2. Program flow instructions to control the execution order of a BASIC program and to permit certain parts to be re-run or bypassed.

3. Expressions containing operations to calculate variables.

4. Input/output statements to regulate the data flow.

5.1 BASIC Commands.

Commands are used to prepare, change or print out a program. To do this, program texts must be stored or loaded, the contents of disks listed and the program started or stopped. In most cases direct mode commands are used. There is a detailed description of all BASIC commands in a later chapter and the following table represents a brief summary:-

BASIC Commands Summary

BLOAD	Load a file from disk.
BSAVE	Save a file to disk.
CONT	Restart an interrupted program.
DELETE	Erase certain program lines.
DIRECTORY	List the contents of a disk.
DLOAD	Load a program from disk.
DSAVE	Saving a program to disk.
HEADER	Format a disk.
LIST	List the program.
LOAD	Load a program from a disk drive or another device.
NEW	Erase the whole program in memory.
RUN	Start the program.
SAVE	Save the program to disk drive or other device.
SCRATCH	Erase a file or program on the disk.
VERIFY	Compare the program in memory with a stored copy.
DCLEAR	Initialise the disk operating system.

5.2 Declaration Statements.

Statements of this type have no direct influence on the running of a program, even if they are executed during a program run. They serve to define certain characteristics which may be used later in the program. An earlier chapter described in detail how the data type of a variable is defined by selecting the last character of a variable name:-

No special character	-	real number
% character	-	integer
$ character	-	text (string)

This definition of the data type represents the simplest form of a declaration statement for the BASIC interpreter. Further statements of this type are shown in the following table:-

Further declaration Statements

DATA defines data tables which can be transferred to variables by using READ statements

DEFFN defines a user function which can be used in later program statements.

DIM variable (index 1.....index n) defines an array variable and reserves space for it.

The DATA statement is discussed in more detail in a later chapter.

Sometimes it is necessary to carry out the same calculation at different points in the program. In such cases it is easier to define this function with a DEF statement at a single point in the program and use the function thus defined as and when required. This saves time in program preparation and uses less memory than if one were to repeat the same calculation over and over again. The DEF statement is explained in detail in a later chapter.

The DIM statement defines data arrays. It is always used when an indexed variable needs more than 11 elements. The use of arrays and DIM statements is discussed in more detail in another chapter.

5.3 Statements for Program Control.

Statements of this type are used either to alter the sequence in which certain parts of the program are processed, or to control some aspect of the computer or program environment. In the absence of special statements, the program will run in a pre-determined sequence dictated by the line numbers. This means program control always goes from one completed program line to the next program line. However, sometimes not all the program lines are to be processed in order. BASIC therefore has a group of statements which allow the continuation of the program from another point.

Sometimes it is necessary to alter certain parts of the program environment. For example the CLR statement can be used to erase all variables. Other statements from this group control the memory.

The program control statements are:-

GOTO	ON...GOSUB
USR	ON...GOTO
CLR	DISPOSE
RESUME	TRAP
END	RESTORE
FOR...NEXT	RETURN
GOSUB	STOP

IF...THEN...ELSE, WAIT, BANK, and SYS are described in a later chapter.

5.3.1 Control of the program run.

BASIC has many statements which determine the sequence in which the individual program parts are to be processed. These statements can be split into three types:-

1. Unconditional statements. The jump in the program is always executed.

2. Conditional statements. The program jump is executed under certain conditions, otherwise no jump occurs and execution carries on undisturbed.

3. Loop statements. A group of instructions are repeated until a pre-condition is met. Then the loop is completed.

Unconditional jump statements.

END	The normal end of program - READY appears on the screen.
GOSUB linenumber	The program is continued at the line whose number is after the GOSUB statement;- used with RETURN to execute a sub-routine before returning to the main body of the program.
GOTO linenumber	The program is continued on the corresponding line;- used to jump over other statements.
RETURN	The program is continued at the statement following the last GOSUB statement;- used with GOSUB to continue in the main body of the program.
STOP	The program is interrupted. BREAK IN linenumber appears on the screen. Subsequently the program may be aborted or continued.

Conditional jump statements.

IF condition THEN linenumber
or: IF condition GOTO linenumber

The program branches only when the given condition is true.

Example: IF a = b THEN 500

The program continues at line 500 only if a and b are equal.

IF condition THEN statement 1: ELSE statement 2

If the given condition is true, statement 1 is executed, otherwise statement 2 is executed.

Example: IF a=b THEN c=d+1: r=sqr(a): ELSE c=d-1: r=sqr(b)

If a=b then c=d+1 and r=sqr(a), otherwise if a is not equal to b, c=d-1 and r=sqr(b)

ON variable GOSUB jumplist

The sub-routine whose position corresponds to the variable in the jump list is called by the GOSUB statement.

Example: ON I GOSUB 100,200,500

If I is 2 then gosub 200 is executed.

ON variable GOTO jumplist

The same as ON GOSUB, but the call is a GOTO.

Loop Statements.

FOR variable = start TO end STEP stepsize...BASIC statement(s)...NEXT variable

All instructions between FOR and NEXT are repeated as a loop. Therefore the variable before the first loop run is set at start. When NEXT is executed, the value step size is added to the loop variable or, if STEP is not given, value 1 (by default). If the variable is still smaller or equal to end, then the whole loop is executed again.

Example: FOR I = 1 TO 10 STEP 2: PRINT A(I): NEXT I

All uneven elements of array a() between 1 and 10 are printed.

WAIT address, mask 1, mask 2

The byte in the address is tested. Firstly the exclusive OR is formed between the contents of the address and the value of mask 2. This intermediate result is ANDed with the value of mask 1. If the result is 0, the WAIT statement is executed again.

Example: WAIT 62255, 1, 1

The program waits at this point until the lowest bit in location 62255 is 0. (If mask 2 is ommitted the default value of 0 is assumed).

Structured programming.

The statements GOSUB, ON...GOSUB and IF...THEN...ELSE form the basis of structured programming. It is possible, using these statements, to divide a large program into small and easily manageable sections.

GOSUB and ON...GOSUB can call a sub-routine from any part in the main program. By using GOSUB and ON...GOSUB statements, a programming problem can be split into several smaller problems which are linked via GOSUB statements. This split clarifies the overall appearance and facilitates debugging when the program is being tested.

The IF...THEN...ELSE statement is one of the most elegant methods of structuring a program. The simplicity and efficiency of this statement saves time and greatly increases the legibility of a program.

5.3.2 Interception of Program Errors.

One of the most important features of the new BASIC 4.Ø + is its capacity to treat errors (bugs) arising in the program. The bug can be trapped, analysed, and the program restarted at the relevant point when suitable changes have been made. The statements TRAP, DISPOSE and RESUME work with the pre-determined variables ER and EL and the function ERR$ (ST, DS and DS$ may also be involved in the handling routines).

Tracking the bug.

The statement TRAP diverts the program to the relevant line. BASICs own treatment of errors (which can still interrupt a program in more complicated cases) is not involved and errors can be treated independently.

Analysis.

The bug treatment routine shows which error has occurred by the variable ER which contains the "error number". The variable EL contains the line number where the error occurred. The text variable ERR$ contains the normal BASIC error message which the computer would otherwise have used in its own error routine. This message can be printed out if required.

Switching off Error Treatment.

A TRAP command without line number parameter reactivates the system's own error treatment. This is of interest if errors have to be trapped only in certain parts of the program, but if the normal error treatment is required otherwise.

Error Treatment and the Stack.

When instructions like GOSUB, ON...GOSUB or FOR are executed, values are placed on the Stack. The DISPOSE statement is used for removing these values. The RESUME statement can continue the program afterwards.

Statements for Error Treatment.

DISPOSE FOR/GOSUB

The Stack entries of a FOR...NEXT loop or a GOSUB...RETURN structure are cleared. Then the program can be continued by using RESUME.

RESUME NEXT/linenumber

When debugging has been completed, RESUME then dictates whether the program carries on at the next statement after the error statement (NEXT) or at any point in the program (linenumber).

TRAP linenumber

When an error occurs the program jumps to the given line number. If the linenumber parameter is not given, then standard error handling is invoked.

Error Messages.
These are accessible using ERR$( ).

0	STOP KEY DETECTED	24	ILLEGAL QUANTITY
1	TOO MANY FILES	25	OVERFLOW
2	FILE OPEN	26	OUT OF MEMORY
3	FILE NOT OPEN	27	UNDEFINED STATEMENT
4	FILE NOT FOUND	28	BAD SUBSCRIPT
5	DEVICE NOT PRESENT	29	REDIMENSIONED ARRAY
6	NOT INPUT FILE	30	DIVISION BY ZERO
7	NOT OUTPUT FILE	31	ILLEGAL DIRECT
8	MISSING FILE NAME	32	TYPE MISMATCH
9	ILLEGAL DEVICE NUMBER	33	STRING TOO LONG
10	ARE YOU SURE?	34	FILE DATA
11	BAD DISK	35	FORMULA TOO COMPLEX
14	BREAK	37	UNDEFINED FUNCTION
15	EXTRA IGNORED	38	?LOAD ERROR
16	REDO FROM START	39	?VERIFY ERROR
20	NEXT WITHOUT FOR	40	OUT OF STACK
21	SYNTAX ERROR	41	UNABLE TO RESUME
22	RETURN WITHOUT GOSUB	42	UNABLE TO DISPOSE
23	OUT OF DATA		

5.3.3 Program environment in BASIC.

There are two statements which alter the environment of a program:-

CLR and RESTORE.

CLR - clears all variables (and resets the Stack)

RESTORE line number - The data pointer to the start of the given line or, if no line number is given, to the start of the first DATA statement in the program.

5.4 Arithmetic Expressions.

Arithmetic expressions are used at many points in a BASIC program. An expression is a combination of variables, constants, function references and operators which produces a single numerical value as a result. For example, A+2. This expression contains variable A, constant 2 and the operator +. The result of this expression is a single numerical value.

5.4.1 Operators.

Operators determine how the variables and constants are related in an expression. There are logical and numerical operators. Logical operators are:-

```
AND (A AND 2)
OR  (A OR 2)
NOT (NOT A) (EOR is not available)
```

NOTE: Logical operations are carried out in 16 Bit binary.

5.4.2 Numerical operators.

Numerical operators are:

+ addition	(A + 2)	
- subtraction	(A - 2)	
* multiplication	(A * 2)	
/ division	(A / 2)	
^ exponentiation	(A ^ 2)	(Do not use ** as an alternative)
digit sign +	(+3)	
digit sign -	(-3)	

Warning:

-2 ^ 2 gives the value -4, and not 4.

The higher valued operator (^) is always executed first and then the lower (-). A different result therefore can be obtained by using brackets:-

(-2) ^ 2 is 4.

NOTE: All arithmetic operations are carried out in floating point format.

5.4.3 Text Operator.

A single operator may be used with text (string) variables. The plus sign + is used to join (concatenate) variables. In this operation text variables are connected so as to form a new text variable. For example:-

```
A$="Text 1"+"Text 2"
```

gives: "Text 1Text 2" as the result in a$.

The length of the resulting string is the sum of the lengths of the individual strings. One must therefore take care that the total length does not exceed 255 characters.

5.4.4 Logical Operators.

If we turn these operators to numbers, then first we should observe the binary presentation. Let us take 35 and 36 as the examples:-

Binary	Decimal
0000000000100011	35
0000000000100100	36

The operation AND now forms the logical AND between both numbers by bit:-

	0000000000100011	35
AND	0000000000100100	36
=	0000000000100000	32

The OR operation works like this:-

	0000000000100011	35
OR	0000000000100100	36
=	0000000000100111	39

In order to understand an IF expression, one must know how the logical values TRUE and FALSE are presented. The logical value TRUE in binary form has 1 in any bit position. The logical value FALSE has 0 in all bit positions.

FALSE = 0000000000000000 = 0

Data expressions are only FALSE if they have a 0 in every position. All other expressions are TRUE.

Therefore, instead of 'IF A<>0 GOTO 21' one could write 'IF A GOTO 21'.

5.4.6 Hierarchy of the Operators.

Individual terms are not necessarily processed in the sequence in which they were entered. Exponentials are evaluated first, then multiplications or divisions and lastly additions or subtractions. Let us examine the simple expression 2+8/2. If this expression were processed in the order in which it is written, 5 would be the result (mathematically incorrect). However, in this example, the division must take place first and then the addition. The correct answer is now 6. Care must therefore be taken when programming formulae. If the formula is to be worked out from left to right, then it should be written (2+8)/2. Brackets (parenthesis) override the normal hierarchy, forcing the expressions in brackets to be evaluated first.

The operators are always carried out in the following sequence:-

1.	^	Exponentiation or "Raising to a power"
2.	* and /	Multiplication and division
3.	+, -, Negation	Addition, subtraction and negation
4.	<,> etc.	Relational Operators
5.	NOT	Logical Operator
6.	AND	" "
7.	OR	" "

Operations at the same level in this hierarchy are evaluated from left to right. So, all arithmetic operations are evaluated first, then the comparisons and finally the logic. To alter this sequence in a formula, brackets must be used. An expression in brackets is always evaluated first. The result of this expression is used in the remaining formula, as in the example above. Bracket expressions can also be nested within each other. In this case, the expression in the innermost brackets is evaluated first. In the expression (A-(B+C))/D, B+C is formed first, the result subtracted from A and then divided by D.

5.4.7 Input/Output Statements in BASIC.

There are a large number of Input / Output (or I/O) statements for:-

- Screen.
- Keyboard.
- Printer.
- Disk drive.
- Serial interface.
- Peripherals on the IEEE bus.

There are two types of I/O statement:-

- Statements for control.
- Statements for data transfer.

BASIC statements used for Data Input/Output.

Control Statements	Transfer Statements
CLOSE	BLOAD
DCLOSE	BSAVE
DOPEN	CMD
OPEN	GET
PUDEF	GET#
	INPUT
	INPUT#
	PRINT
	PRINT#
	PRINT USING
	PRINT# USING

BLOAD and BSAVE are dealt with in detail in a later chapter. The I/O statement represent a bridge between the program and the outside world. Without these commands the program can still alter data but it is unable to present results. If you need to read data stored in external memory, the computer must first be told the storage location (on which device) and then the name of the storage file. Likewise, for storing the system must know under which name to store the data, and on which device.

5.5.1 Preparation of data Input/Output.

The control statements are used to prepare the system for data transfer and to open or close channels to the corresponding peripherals.

The commands OPEN and CLOSE are used to:

- Allocate a file or peripheral with a channel number.
- Open a file.
- Close the file after data transfer.
- Activate a device such as a printer.

Preparation for Data transfer statements.

OPEN channelnumber, peripheralnumber, (command), (openingtext).

Open a data channel for a peripheral device and allocate a logical channel number. Several commands can be given to the device, and an opening text may also be sent, depending on the device and file type.

CLOSE channelnumber

This closes all I/O operations for the channel which was given this channel number.

NOTE: Before giving any commands to transfer data from a file to the computer memory, the peripheral must first have a channel number assigned to it. This channel number will be used in all data transfer statements to tell the system where the data should go or where it can be obtained. Some devices recognise certain special commands. For example, one can tell a printer to move the paper to the top of the next page. Once a file has been opened, program control enables you to read from it or write to it. If a device or file is no longer needed, the channel should be closed. If the CLOSE command is not given, data may be subsequently lost or corrupted.

5.5.2 Data Transfer.

After establishing the channel, data transfer can be executed using BASIC statements. Some transfer commands serve to obtain information for the program from the user. Others tell the user what the program is doing. For example, the INPUT command is used to gain information from the keyboard and the INPUT# command to get information from a file. The PRINT command gives the user results, the PRINT# command sends data to a file.

Input/Output Statements.

BLOAD filename ON Bbank, P offset

Reads binary information from a file and stores it in the memory segment bank starting at location offset. BLOAD reads a file as binary data and not as program text.

BSAVE filename ON Bbank, P start TOP end

Copies the memory contents from the segment bank in the area between start and end to the file specified in filename.

CMD Channelnumber (,text)

Output, usually to the screen, is switched to the channel number by this command. A text can be sent and appears as the first line output. The device is left 'listening'.

GET Variable

Reads a single character from the keyboard. GET does not wait for input. If the keyboard buffer has no more text characters, the program will run on and the variable will be assigned 0 or null as appropriate.

GET# Channelnumber, variable

Reads a single character from the channel and allocates it to the variable. This command does not wait if there is no character to read.

INPUT (promptstring), variablelist

Prints the promptstring on the screen and waits for input from the keyboard. This data is then transferred to the variable(s) in the list. If each variable has not been given a value, a double question mark in printed and the input for the next variable requested. The program waits until all variables have an acceptable value.

INPUT# Channelnumber, variablelist

Reads data from the channel and allocates them to variables in the variable list until all variables have a value. The program is interrupted for as long as this takes.

PRINT (Variablelist)

Prints all variables, expressions and functions from the variablelist to the current output device, usually the screen. PRINT uses standard BASIC formatting.

PRINT# Channelnumber, Variablelist

Writes the variablelist to the channel.

PRINT USING Formatlist, Variablelist

Gives formatted data output on the current output device. The print format is defined by the formatlist.

PRINT# Channelnumber USING formatlist, variablelist

Formatted output to a channel.

PUDEF Controltext

Defines full characters, separation characters, decimal point symbols and currency characters by the characters which have been given to the controltext. These characters are used in the format output by PRINT USING.

READ Variablelist

Reads data from lines in the program.

5.4.4 Relations.

These are operators which compare two values with one another. These are:-

<	smaller than
<=	smaller than or equal to
>	larger than
>=	larger than or equal to
=	equal
<>	Unequal

An expression which uses comparative operators can only have a TRUE or FALSE result. For example: A>B tests if the value of A is larger than that of B. These operators are mainly used in connection with the IF statement. A typical example:-

```
IF (A>B) OR (C<D) GOTO 1000
```

In this case the expressions A>B and C<D are connected by the logical operator OR. There are two conditions of which at least one must be TRUE. There is then a jump to line 1000.

Logical expressions in BASIC.

If logical operators appear in an equation, the numerical values of the variables in question are converted to the 16 Bit binary format. The individual logical operations are then executed by bit. The value 35, for example is presented as 0000000000100011 in the binary format. Logical operations are AND, OR, and NOT. The first two operate on two numbers and NOT operates on a single number. The AND operator only produces a 1 if both variable values connected by it were logically 1 also:-

1 AND 1 is 1
0 AND 1 is 0
1 AND 0 is 0
0 AND 0 is 0

The OR operator produces a 1 if either of the values was a logical 1:-

1 OR 1 is 1
0 OR 1 is 1
1 OR 0 is 1
0 OR 0 is 0

CHAPTER 6

SOUND AND MUSIC

Introduction

Tone production with your computer has three main uses: playing of musical pieces, producing sound effects, and the sounding of 'warning noises'.

6.1 Structuring a Music Program

The sound of a tone is determined by four characteristics:

Pitch, volume, waveform and envelope. The last two of these enable us to differentiate between various instruments by ear and these characteristics will also need to be influenced in your program.

Your 700 has for this reason a special integrated circuit: The Sound Interface Device (SID). The SID has a range of memory locations reserved for parameters which control the synthesis of a desired sound. You already know that your 700 can simultaneously produce three voices. Let us consider the first of these. The base address of the SID is 55808 in memory bank 15, (the system bank). (E.g: SI = 55808 assigns the base address to the variable SI).

The pitch is physically determined by the frequency. The frequency is stored by a parameter in the SID, and this can assume values between almost 0 and 65000. As it is impossible to store such large numbers in a single memory location, we must break down the frequency parameter in to one high and one low byte. These bytes occupy the first two registers of the SID:-

```
FL = SI     (frequency, Lo-byte)  :REGISTER 0 is the 1st register.

FH = SI+1   (frequency, Hi-byte)  :REGISTER 1 is the 2nd register.
```

16 settings are allowed in the SID for the volume - from 0 (switched off) to 15 (full volume). The corresponding parameter is stored in Register 24:-

```
L = SI+24      (volume)  :REGISTER 24 is the 25th register.
```

Now comes the waveform. The SID offers four fundamental forms: triangle, sawtooth, square and noise. Each one is controlled by a bit in Register 4:-

W = SI + 4 (waveform)

In order to select one of the waveforms, you write into this register one of the parameters 17, 33, 65 and 129. If you choose 65 (square wave) you must also determine a futher parameter between 0 and 4095 for the pulse width. The two bytes of this parameter are in registers 2 and 3:-

TL = SI+2 (pulse width, Lo-byte)

TH = SI+3 (pulse width, Hi-byte)

Finally, we have the 'envelope'. Your 700 allows every tone to rise to the volume set in register 24 - then to decay somewhat - the volume now stays fixed as long as you keep the tone switched on.

Then the volume subsides. Four parameters take part in this envelope which the SID processes in 2 further registers:-

A = SI+5 (attack and decay)

H = SI+6 (sustain and release)

Each one of these registers is split into two: the parameter in the 4 higher bits from A determines the rise time of the tone and the parameter in the 4 lower bits determines the decay. Small values mean quick/hard; large values mean slow/soft. This also applies to the lower 4 bits of H which control the fade of the tone after switching off. The 4 higher bits of H determine the volume at which the tone is held (sustain level)- the highest value gives the volume previously set in register 24, lower values cut this volume proportionately.

6.2 Sample Program

You must first decide which voices (or tone generators) you want to use. For each of these voices, the settings (volume, waveform, etc.) must be determined. You can use up to three voices simultaneously - this example uses only voice one:-

```
10 SI+55808:FL=SI:FH=SI+1:W=SI+4:A=SI+5:H=SI+6:L=SI+24:REM DEFINE

20 BANK 15                      :REM SID is in bank 15

30 POKE L,15                    :REM Full volume

40 POKE A,16+9:POKE H,4*16+4    :REM ADSR

50 POKE FH,14:POKE FL,106       :REM Hi and Lo byte of the frequency

60 POKE W,17                    :REM Waveform. (Should always be set
                                 last since the lowest bit in this
                                 register switches the tone
                                 generator on or off.)

70 FORT=1TO500:NEXT             :REM Loop to set duration of tone

80 POKE W,0:POKE A,0:POKE H,0   :REM Switch off.
```

Type RUN to hear the sound generated by this program (The REMs may be omitted).

6.3 Melodies

You don't have to be a musician to produce melodies with your 700.

Here is a sample program which shows how it is done. We are using only one of the three available voices. Erase or save the previous program and try the following:-

```
10 SI=55808:FL=SI:FH=SI+1:W=SI+4:A=SI+5:H=SI+6:L=SI+24:REM
Definiton of register addresses

20 BANK 15                    :REM SID is in bank 15.

30 POKE L,15                  :REM Full volume.

40 POKE A,9                   :REM Attack/Decay.

50 READ X:READ Y              :REM Hi-byte lo-byte of the frequency
                               from the data lines in 130 and 140.

60 IFY=-1THENPOKE W,0:END     :REM (When the program finds the -1 at
                               the end, it will switch off.)

70 POKE FH,X:POKE FL,Y        :REM Set frequency.

80 POKE W,17                  :REM Set waveform and switch on.

90 FORT=1TO100:NEXT           :REM Tone duration (delay loop).

100 POKE W,0                  :REM Switch off.

110 FORT=1TO10:NEXT           :REM Short pause to fade.

120 GOTO 40                   :REM Next sound.

130 DATA8,146,9,159,10,205,11,113,12,216,14,106,16,46,17,37

140 DATA-1,-1                 :REM These data (useless as frequency)
                               end the program in line 60.
```

The numbers in the data statements in line 130 are pairs, each representing the hi-byte and lo-byte of the C-sharp scale.

If we want to produce tones which are similar to those from cymbals, we must alter line 80 in the following way:-

```
POKE W,33
```

By using this POKE command, we are selecting a sawtooth waveform; this means that we obtain "sharper" sounds than in the triangular waveform used previously.

But selecting the waveform is only one of the ways to determine the sound character. We can turn the cymbals into a banjo by altering the choice of the attack/decay value. This can be done by using the following command in line 40:-

```
POKE A,3
```

In this way, you can imitate the sound of various instruments.

6.4 Other Sound Settings

6.4.1. VOLUME

Selection of volume is made for all three tone generators simultaneously. The register for this has the address 55832. Maximum volume is attained by poking 15 into this register:

POKE L,15 or POKE 55832,15

To turn off the tone generators, put a 0 in the register:-

POKE L,0 or POKE 55832,0

The volume is generally set at the beginning of a music program; but interesting effects may be achieved by programmed alteration of the volume.

6.4.2. WAVEFORM

As seen in our example, the waveform largely determines the character of a sound. You can set the waveform separately for each voice - you have a choice between triangle, sawtooth, square and noise.

The following table gives a summary:

Summary of waveform setting

Voice	Location	Waveform	Value
		Square	65
1	4	Sawtooth	33
2	11	Triangle	17
3	18	Noise	129

Thus POKE 55808+11,17 sets voice 2 to use the Triangle waveform. (Remember 55808 is the base address of the SID).

6.4.3 ENVELOPE

The values for attack and decay (which can be selected separately for each voice) are used together as a single value. The attack parameter gives the time it takes for the tone to reach its (predetermined) volume, the decay parameter is a measure of how quickly the volume decays to the sustain level. If 0 was selected as the sustain level, then the decay parameter gives the release time (to volume 0) and thus determines the length of tone. The address for the individual voices and the values corresponding to the various settings can be seen in the following table. (The values selected for attack and decay are added and the sum POKEd into the corresponding register.)

Attack/Decay setting

Voice	Location	
1	5	Attack value ranges from 0 to 240 in
2	12	multiples of 16.
3	19	Decay values range from 0 to 15.

Thus POKE 55808+12,(16*2)+13 sets voice 2 to a fairly hard attack and a fairly soft decay.

The following program is a further example of these commands in use:

```
10 REM 6.4.3
20 SI=55808:FL=SI:FH=SI+1:TL=SI+2:TH=SI+3:W=SI+4:A=SI+5:H=SI+6:L=SI+24
30 PRINT"PRESS A KEY"         :REM Screen message.
40 GETZ$:IFZ$=""THEN40        :REM Wait for Key.
50 BANK15:POKE L,15           :REM Volume.
60 POKE A,1*16+5              :REM Attack and decay.
70 POKE H,0*16+0              :REM Sustain and release.
80 POKE TH,8:POKE TL,0        :REM Pulse width.
90 POKE FH,7:POKE FL,53       :REM Frequency.
100 POKE W,17                 :REM Waveform, generator on.
110 FORT=1TO200:NEXT          :REM Duration.
120 POKE W,0                  :REM Off.
130 GOTO40                    :REM Repeat.
```

Voice 1 produces a tone with short rise time and short decay phase when the maximum volume has been reached, (line 60). What can be heard should sound like a ball which is bouncing about inside a lead drum. To produce another sound, we must alter this line.

Stop the program with RUN/STOP. List the program and alter line 60 as follows:

```
60  POKE A,11*16+14
```

The tone produced with this new setting sounds something like an oboe or some other woodwind instrument.

Experiment yourself, change the waveform and envelope to get the feeling of how the various values of these parameters can change the character of the tone.

Similar to the previous register, the sustain and release of the sound are determined by a numerical value which can be calculated by adding the values which appear in the following table:

Sustain/Release setting

Voice	Location	
1	6	Sustain value ranges from 0 to 240 in
2	13	multiples of 16.
3	20	Release values range from 0 to 15

Thus POKE 55808+13,(16*2)+13 sets voice 2 to a fairly quiet sustain level and a fairly slow release.

Change the 0's in line 70 to any value up to a maximum of 15 and listen to what emerges!

6.4.4 THE CHOICE OF VOICE AND NOTES

As already stated, to produce a tone, you must use two values for the frequency. Because the voices are controlled by different registers you can independently program the three SID voices and, for example, produce a three-voiced piece of music.

POKE values for the middle octave

	Location				Value												
Voice	1	2	3	Note	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C
Hi-byte	1	8	15		17	18	19	20	21	22	24	25	27	28	30	32	34
Lo-byte	0	7	14		37	41	62	90	153	226	62	175	54	213	139	92	73

To generate 'C' with voice 1, you must use the following commands:

```
BANK 15: POKE 55809,17: POKE 55808,37
```

or `POKE SI+1,17:POKE SI,37`

The same tone with voice 2 can be obtained by:

```
BANK 15:POKE 55816,17:POKE 55815,37
```

or `POKE SI+8,17:POKE SI+7,37`

Sound Effects

Unlike music, sound effects should accentuate events on the screen (explosion of a space ship, etc.) or they should inform or warn the user of a program. (For example, that he is in the middle of erasing his data disk.)

Here are a few suggestion for experimentation:-

1. Alter the volume during the tone to produce an echo effect.
2. Jump quickly from one sound level to another, to achieve tremolo.
3. Try out the different waveforms.
4. Study the envelope. (Ask a synthesizer player about ADSR.)
5. Surprising effects can be obtained by varying the programming of the three voices (eg: hold the tone in one voice for longer than in another).
6. Use the square wave and change the pulse width.
7. Experiment with the noise generator to produce explosion noises, arms fire, footsteps, etc.
8. Alter the frequency quickly over several octaves.
9. Use a frequency setting that alters.

CHAPTER 7

BASIC COMMANDS

INTRODUCTION

The following chapter describes in detail all commands for the BASIC 4.0+ interpreter. The special commands for disk use, such as HEADER, SCRATCH, COPY, etc. are each described in the user's manual for the floppy disk.

BASIC commands are used to change, run, start or erase a program. When the command is executed depends on whether it is entered in direct mode (without line number) or in program mode (with line number as part of a BASIC program).

Commands in direct mode are executed as soon as the RETURN key has been pressed. Commands in the program mode are executed just as BASIC statements, when it is their "turn" in the program. The CONT command cannot be used in a BASIC program. This section deals with the following commands:-

CONT	DLOAD	NEW
DELETE	DSAVE	RUN
DIRECTORY	LIST	

7.1 CONT

Format: CONT

Abbreviation: cO

The CONT command is used to start a program again after an interruption. The reason for the interruption may be:-

- The STOP key was pressed
- The program executed a STOP statement
- The program executed an END statement

When CONT has been entered, the program runs on from the point it was interrupted. If the program is interrupted, the actual value of the variables can be examined, variable value altered or a list made on the screen. This command is very useful, therefore, for debugging.

CONT does not function if:-

- The program itself was altered.
- The program has stopped because of an error.
- An error has occurred during the interruption by use of commands or statements in the direct mode.

If the CONT command cannot restart the program, the error message:

?CANNOT CONTINUE

appears.

7.2 DELETE

Format: DELETE [from] [-] [to]

Arguments: from gives the line number of the first BASIC statement which is to be erased to is the number of the last BASIC line to be erased.

Default: (if nothing is given)

from = first line of BASIC program
to = last line of BASIC program

Abbreviation: dE

The DELETE command is used to erase one or more program lines from the program memory. It erases all lines between from and to inclusive. If only one argument is given (from) , then only one single line is erased. If both are left out but the dash given, then the whole program in the memory is erased. Examples:-

DELETE 20-50	erases lines 20 to 50
DELETE -75	erases all lines from program start to line 75
DELETE 300-	erases all lines from 300 to end of program
DELETE -	erases the whole program.
DELETE 0	erases the whole program
DELETE	by itself generates a syntax error.

7.3 DIRECTORY

Format: DIRECTORY [Dnumber] [, filename] [, Uaddress]

Arguments: Dnumber is the drive number, whose contents are to be presented.
filename is the name of a data file, always in inverted commas or as a string variable in brackets. (The name may also contain the special characters "*" or "?" to pattern match the name, or "=p/u/r/s" to pattern match the file type.)
address is the device address of the memory unit on

the IEEE bus (usually 8).

Default: If the parameter is not given, the contents of the disks in both drives are shown on the screen. If no filename is given, all disk files are fetched. Without address device 8 is assumed.

Abbreviation: diR

The DIRECTORY command fetches a list of all data files which have been put on to disk.

If a star (*) is used as last character of a filename, only those filenames will appear on the screen which correspond with the letters in the filename up to the star. If a question mark is used within the filename, then all filenames will appear corresponding to the rest of the filename. Example:-

DIRECTORY "test??data"

A list will be fetched with all filenames which have the letters "test" and "data" at the given points, eg:-

"test01data"

"testxydata"

"test..data", etc.

ie. "?" means that there must be a character in the filename, but it may be any character.

The star is used to ignore the rest of the filename.

Example:-

DIRECTORY "test*"

A list will be fetched with all the filenames which start with the letters "test", eg:-

"test0a data"

"testdata"

"test program"

"testscorecard"

"test", etc.

The use of star or question mark can present parts of the disk contents in one easy-to-survey manner. Other examples:-

DIRECTORY	The filenames of all files on both disk drives are fetched.
DIRECTORY "pgm#*"	Names of all files which start with "pgm#" are fetched.
DIRECTORY d0,"DATA*"	The names of all files which begin with the "DATA" in drive 0 are fetched.
DIRECTORY "??xyz"	The names of all files which have any two characters in position 1 and 2, followed by letters "xyz" are fetched.
DIRECTORY "*=S",dl,U9	Fetches all sequential files of any name from drive 1 of unit 9. Note: diRu9,dl,"*=S" would serve the same purpose.
DIRECTORY (A$)	Fetches files whose names or types correspond to whatever A$ is assigned.

7.4 DLOAD

Format: DLOAD filename [,Dnumber] [,Uaddress]

Arguments: filename is the name of the file which is to be loaded. The name can either be directly given, or can be a text variable. If the name is directly given, it must be within inverted commas. If a text variable is given, it must be within brackets.

Default: number = 0 : drive
address = 8 : unit

Abbreviation: dL

The DLOAD command is used to load BASIC programs stored on a disk into the program memory. (BLOAD command must be used for other files.) DLOAD can be used to load BASIC programs from older Commodore computers. To store a program with DSAVE and load it onto an older Commodore computer is, however, only possible with a special preliminary procedure or (see Technote 500/700-014) an auxiliary program. DLOAD can also be used during a program. When the DLOAD has been executed, the new program is started immediately. The variables of the old program are retained (or may be erased with the CLR command).

Example:

Store the program called "ONE" with DSAVE"ONE" on your disk in drive 0, then enter the program called "TWO" and start with RUN:

PROGRAM TWO:

```
1ØØ  REM TWO
11Ø  REM
12Ø  REM CALL UP PROGRAM
13Ø  REM
14Ø  REM HERE THE VALUES
15Ø  REM OF THE VARIABLES ARE DEFINED
16Ø  REM
17Ø  A=1ØØ
18Ø  A$="FRED BLOGGS"
19Ø  DLOAD"ONE"
2ØØ  REM THIS LINE IS NEVER REACHED
21Ø  PRINT "IF YOU SEE THIS, THERE HAS BEEN ERROR"
22Ø  END
```

PROGRAM ONE:

```
1ØØ  REM ONE
11Ø  REM
12Ø  REM THIS PROGRAM READS THE VARIABLES
13Ø  REM OF THE CALLING PROGRAM
14Ø  REM
15Ø  PRINT A$" IS"A" YEARS OLD"
16Ø  END
```

7.5 DSAVE

Format: DSAVE filename [,Dnumber] [,Uaddress]

Arguments: filename is the name of the file which is to be stored by DSAVE. The name can be given directly or can be in a text variable. If it is given directly, it must be enclosed within inverted commas; if a text variable is given, it must be in brackets.

Default number = Ø
address = 8

Abbreviation: dS

The command DS\VE is used to store programs on a disk. DSAVE can also be used within a program. It is often necessary to update the program copy on the disk. If the new program version is to be stored on disk under the same name, the old disk file must first be erased. To do this, the special sign "@" can be written at the start of the data file name.

Example: DSAVE "@june"

By this command, the program is written from the memory to the file "june". The old contents of data file "june" will therefore be replaced. This is known as "save-with-replace".

7.6 LIST

Format: LIST [from] [-] [to]

Arguments: from gives the line number of the first BASIC statement to be listed.
to is the number of the last BASIC line to be listed.
the '-' must be included if more than one line is to be listed and from or to are specified.

Default: from = first line in BASIC program
to = last line of the BASIC program

Abbreviation: lI

The LIST command is used to display one or more program lines on the screen. The command displays all lines between from and to inclusive. If only one argument (from) is used, then only one line is listed on the screen. If both are omitted, the whole program will be listed on the screen.

Examples:

LIST 200	displays line 200 only.
LIST 20-50	displays lines 20 to 50
LIST - 75	displays all lines from start of program to line 75
LIST 300	displays all lines from 300 to program end.
LIST	displays the whole program

If the program is longer than 25 lines, the screen automatically "scrolls" upwards. (Use C= to stop and CTRL to slow the scroll.)

Program alterations are easily executed with the LIST command. The program line to he altered is first displayed on the screen by LIST. Then the cursor is used to reach the point which is to be altered. The BASIC program text can now be altered. Afterwards, by pressing the RETURN key, the computer makes this alteration in its program memory.

The program is only changed in the memory and not on any copies which may be on disk. If the alteration is also to be carried out on disk or cassette, the program must be stored again with DSAVE.

7.7 NEW

Format: NEW

Abbreviation: None

The NEW command is used to erase a BASIC program and all its data from the memory of the computer.

It will not affect the disk. The NEW command can also be used within a program to erase the program after processing.

7.8 RUN

Format: RUN [linenumber]

Arguments: Linenumber is the number of the line where the program is to start.

Default: linenumber = first line of the BASIC program

Abbreviation: rU

The RUN command starts a BASIC program which is in the program memory.

All variables are first cleared and then program control moves to the program line whose number is given in the linenumber argument. If this argument is not given, the run starts with the first line of the program.

When a linenumber has been given, but the line does not exist in the BASIC program, the error message:

?UNDEFINED STATEMENT

appears on the screen.

The RUN command can also be used within the program itself. It must, however, be noted that all variables will be cleared before the new start.

CHAPTER 8

BASIC STATEMENTS

BASIC statements alter data, variables, memory and the program flow.

BASIC statements may be divided as follows:-

- Declarations/allocations
- Input/Output
- Program control
- Loop control
- Conditional branching
- Unconditional branching

Below is a summary of all BASIC statements which will be individually described in this chapter. Special statements for the floppy disk such as HEADER, SCRATCH etc, are not dealt with here and are explained in the floppy disk manual.

BASIC Statements

Statement Type:-	Declaration/ Allocation	Input/ output	Program control	Branches Cond.	Branches Uncond.	Loop control
CLOSE		X				
CLR			X			
CMD		X				
DATA	X					
DEF FN	X					
DIM	X					
DISPOSE			X			
END			X		X	
FOR..TO..STEP			X			X
GET		X				
GET#		X				
GOSUB			X		X	
GO TO, GOTO			X		X	
IF...THEN...ELSE			X	X		
IF...GOTO			X	X		
INPUT		X				
INPUT#		X				
LET**						
NEXT			X			
ON...GOSUB			X	X		
ON...GOTO			X	X		
OPEN		X				
POKE***						
PRINT		X				
PRINT#		X				
PRINT USING		X				
PRINT# USING		X				
PUDEF		X				
READ		X				
REM			X			
RESTORE			X			
RESUME			X			
RETURN			X		X	
STOP			X		X	
SYS			X		X	
TRAP			X			
WAIT			X			X

**LET is the key word for a value allocation. The word LET, however, need not be used.

***POKE is a special form of allocation which is described in detail in a later chapter.

Most of the BASIC statements can be used in direct mode in a similar manner to BASIC commands. If a BASIC statement without a line number is given, it will be executed as soon as the RETURN key is pressed.

Direct mode execution is useful, for example, to establish the present value of a variable:-

```
?A%,X
```

Direct mode can also be used to operate the computer as a pocket calculator:-

```
?(45.6*19.88)/(SQR(500)*0.85)
```

However, some BASIC statements such as GET, cannot be used in direct mode. If an attempt to do so is made, the error message ?ILLEGAL DIRECT will appear.

Every BASIC statement to be used in the program mode must be in a line which starts with a line number. If several statements are placed on the same line, they are separated by colons (:). In this case, the linenumber is only at the beginning of the line.

The format data in this chapter contain a line number parameter which must always be given if the statements are to be used in the program mode. Line numbers are integers in the range 0-63999.

8.1 BANK

Format: Line number BANK expression

Arguments: expression is a numerical expression or a variable with a value between 0 and 15.

Default: None: BANK by itself generates a syntax error.

Abbreviation: baN

The BANK command defines the memory bank with which some BASIC statements and functions (such as PEEK, POKE, BSAVE) work. The memory is divided into 16 banks each containing 64K. The BANK command will define the bank which will be used by the CPU as data area during a special indirect indexed memory call.

If a program is started with RUN, this is set at 15.

Example:

```
1Ø   REM store the value 2Ø in address 1Ø24 of bank 2
2Ø   BANK 2
3Ø   POKE 1Ø24,2Ø
```

8.2 BLOAD

Format: linenumber BLOAD filename
[, ON Bbanknumber] [,Poffset]

Arguments: filename is the name of the data file which is to be loaded and can either be a text (in inverted commas), or a text variable (in brackets).
Banknumber shows which memory bank the file is to be loaded into.
Offset gives the start address for the load within the bank.

Defaults: banknumber = 15 or the number of the last memory bank selected by a BANK statement.
offset = address from which it was saved.

Abbreviation: bL

The BLOAD statement loads a binary file at any point in memory. Each BLOAD statement can only load into a single memory BANK. Several BLOAD statements must be combined to load information which exceeds a bank boundary in memory (or the Machine Code Monitor may be used). If the Banknumber argument is not given, the information is loaded either into bank 15 or into the bank selected by the last BANK statement. It must be remembered that the addresses Ø and 1 of each bank are reserved for system purposes. Thus, no data should be loaded in these locations. (The offset parameter should therefore always be larger than 1.)

Example:

```
1ØØ  BLOAD "SUB1",DØ,ON B2,P1Ø24
```

The data file SUB1 is loaded from drive Ø into BANK 2 from 1Ø24. Afterwards, unlike the DLOAD command, the program continues with the next BASIC command.

8.3 BSAVE

Format: linenumber BSAVE file name

[,ONBbanknumber] [,Pstartaddress] [TOPendaddress]

Arguments: filename is the name of the file which is to be stored and can either be a text (in inverted commas), or a text variable (in brackets).
banknumber shows from which memory bank the program is to be stored.
startaddress: start address.
endaddress: end address.

Defaults: banknumber = 15 or the number of the last storage bank selected by a BANK statement.
startaddress = 65535
endaddress = start address

Abbreviation: bS

The BSAVE statement stores binary files on to a disk from anywhere in memory.

Each BSAVE statement can only store from one single memory BANK, so several BSAVE statements must be combined in order to store information which exceeds a bank boundary. If the bank number parameter is not given, the information will be stored from the bank selected by the last bank statement.

Example:

```
100  BSAVE"SUB1",D0,ON B2,P1024TOP2048
```

The memory location 1024-2048 in Bank 2 is stored on drive 0 in the datafile "subl".

8.4 CLOSE

Format: linenumber CLOSE channelnumber

Arguments: channelnumber = number of the Input/Output channel which is to be closed.

Defaults: None.

Abbreviation: clO.

The CLOSE statement closes a channel previously opened by OPEN.

All data for this channel still in the memory is first transferred to the peripheral. Thus, the channel is freed for further use by an OPEN statement.

```
100  OPEN 6,4: REM 6 IS THE CHANNEL NUMBER
...BASIC statements...
210  PRINT# 6,A$,B%
...BASIC statements...
550  CLOSE 6
```

8.5 CLR

Format: linenumber CLR

Abbreviation: cL

The CLR statement erases all variable values from the memory. The individual actions are:-

- All numerical variables are returned to 0
- All text variables are erased
- All arrays are erased (any DIM statements are 'cleared')
- Memory pointers are reset.
- System STACK is cleared.

Therefore, care must be taken in a BASIC program to avoid any errors by the misuse of the CLR statement. If, for example, the CLR statement is used with a subprogram, the ensuing RETURN command is no longer able to jump back from the sub to the main program as the stack no longer contains a return address.

The CLR statement is useful to start a new program. (The instructions RUN and NEW execute a CLR as part of their own execution.)

8.6 CMD

Format: linenumber CMD channelnumber [,text]

Arguments: channelnumber is the number of a channel previously opened for a peripheral by OPEN or DOPEN.
text is text (in inverted commas), a text variable or numerical expression which is written to the channel by the CMD statement.

Abbreviation: cM

By using this statement, the information which normally goes to the screen is diverted to a predetermined channel. It can therefore be used to list a program to the printer. Before the CMD can be used, OPEN or DOPEN must first open a corresponding channel. To end the CMD and restore standard output, the PRINT# statement, followed by a CLOSE statement for the relevant channel is used.

CMD statement sequence is as follows:-

```
10  OPEN 6,4 :REM SET UP CHANNEL
20  CMD 6 :REM DIVERT DEFAULT OUTPUT FROM SCREEN TO CHANNEL 6
30  PRINT A;B;C;A$ :PRINT B$ :REM SEND DATA
40  PRINT#6 :REM 'UNLISTEN' CHANNEL 6
50  CLOSE 6 :REM CLOSE CHANNEL
```

By giving these statements in direct mode the values of A,B,C,A$ and B$ are printed instead of being displayed on the screen. By using the CMD statement in a program, (as above) the total output which normally would have appeared on the screen by the PRINT statement can be diverted. (To the printer for example.)

8.7 DATA

Format: linenumber DATA constant [,constant,...,constant]

Arguments: constant is either a text or number which is to be read by a READ statement.

Abbreviation: dA

A data statement is not executable. It is used together with the READ statement. There can be as many arguments on a DATA statement as there is space for them in a single program line. If more constants are needed than fit into a single DATA statement, a new DATA line is begun until all are defined. Care must be taken to place data in the order in which they are to be read.

RESTORE enables single DATA statements to be processed repeatedly by READ. RESTORE is used to indicate which DATA line should be used in the next READ statement. If text constants contain the special characters (for example, comma or semicolon) the whole text must be enclosed within inverted commas. Example:

```
10   DATA fred,janet,3,2.4,"a,b,c"
20   READ A$,B$,xl,y,C$
30   READ A%,D$
40   PRINT A$,B$,xl;y
50   PRINT C$,A%,D$
60   DATA 4711,"this is a text"
```

Result:

```
fred     janet      3 2.4

a,b,c    4711       this is a text
```

8.8 DEF FN

Format: linenumber DEF FNname(argument) = expression

Arguments: name is a valid variable name which is used here as function name
argument is a dummy variable which may later be used to transfer a variable to the function when it is used.
expression is the equation to calculate the desired function.

Abbreviation: dE

This statement allows the user to define his own numerical functions. The expression indicates how the function value is to be calculated. When function is called, the dummy variable (used in the definition) is replaced by the argument in the function call. Example:

```
150  DEF FNAB(X)=X+Y
160  Y=100.5
170  Z=55.8
180  Q=FNAB(Z)
190  PRINT Q
```

The result of this program is 156.3, the sum of Y and Z. The parameter Z became the actual argument of this function in line 180, despite the use of X as a dummy when the function was defined in line 150. Functions can be used in an arithmetic expression just like the built-in BASIC functions or variables. Integer functions or text functions are not definable. All calculating rules for real number evaluation must therefore also be used for defined functions.

8.9 DIM

Format: linenumber DIM variable (index[,...,index]) [,variable (index[,...,index])]

Arguments: variable is a valid BASIC variable name for any type of variable.
index is an expression or a variable which is used as an integer to define the size of the array.

Defaults None: Without DIM, DIM(10) is assumed when the array is first used.

Abbreviation: dI

The DIM statement reserves memory space for arrays. The maximum size of the arrays is determined by size and number of the indices. All indices start at 0 and end at the maximum value given in the DIM statement - an index must not exceed 32767, however. The number of indices depends on how many dimensions the array should have. (A maximum of 255 indices may be specified, though this is not really practicable.)

Example:

A(5) is an array with 6 elements (0,1,2,3,4,5)

B(120,9) - consists of 1210 elements (121*10)

C$(5,5,5) has 216 elements (6*6*6)

Care must be taken when dimensioning arrays not to exceed the maximum available memory space for variables. During dimensioning all array elements are set at 0 or null. The following example shows the application of the DIM statement:

```
10   DIM A(5),B%(2,3)
15   DIM C$(100)
20   DATA 0.0,1.1,2.2,3.3,4.4,5.5
30   DATA 0,1,2,3,4,5,6,7,8,9,10,11
40   FOR I = 0TO5
50   READ A(I)
60   NEXT I
70   FOR J = 0TO2
80   FOR K = 0TO3
90   READ B%(J,K)
100  NEXT K:NEXT J
110  FOR L = 0TO100
120  C$(L)="AAAA"
130  NEXT L
140  PRINT "ARRAY A CONTAINS:"
150  FOR I = 1TO5
160  PRINTI,A(I)
170  NEXT I
180  PRINT "ARRAY B% CONTAINS:"
190  FOR J = 0TO2
200  FOR K = 0TO3
210  PRINTJ,K,B%(J,K)
220  NEXTK:NEXTJ
230  PRINT "ARRAY C$ CONTAINS:"
240  FOR I = 0TO100
250  PRINTI,C$(I)
260  NEXT I
270  END
```

8.10 DISPOSE

Format: linenumber DISPOSE [FOR/GOSUB]

Abbreviation: diS fO/diS goS

DISPOSE is used, together with TRAP, for debugging (error treatment) DISPOSE manipulates the BASIC stack. If the error has occurred in a subprogram or in a FOR...NEXT loop and if the program must continue outside the loop or subprogram after dealing with the error, then information must be removed from the stack which would have been processed by the NEXT statement or RETURN. When the system stack has been corrected, the program can continue. DISPOSE cannot be executed in direct mode. For example:

A program is executing a FOR...NEXT loop. During this, a division by 0 occurs, which is trapped by the TRAP statememt:

```
10    TRAP 1000
...BASIC PROGRAM...
120   FOR I = 1TO100
130   A=I/B :REM error since b=0
140   NEXT I
150   PRINTA :END
1000  REM error treatment
...error analysis...
1100  DISPOSE FOR:REM removes the loop from the STACK
1110  RESUME 150
```

8.11 END

Format: linenumber END

Abbreviation: eN

The END statement ends the current program. The content of all variables is unaltered. READY appears on the screen. The program may be restarted by CONT. END need not be given as the last program statement. It can be omitted or taken at any point within the program. END is not illegal in direct mode, but is rather pointless.

8.12 FOR

Format: linenumber FOR variable = expression1 to expression2 [STEP expression3]

Arguments: variable is a real variable which is changed with every loop run.

expression1 is a variable or an arithmetic expression to preset the initial value of a variable.

expression2 is a variable or an arithmetic expression which ends the loop processing if the variable exceeds this value.

expression3 is a variable or an arithmetic expression which is added to the value of the variable during every loop run.

Defaults: expression3=1

Abbreviation: fO

The FOR and associated NEXT statements define a program loop. The loop variable initially assumes the value of expression 1: all statements belonging to this loop are processed as far as the NEXT statement. When this is reached, the value of expression3 is added, or (if no STEP parameter is given) 1, to the loop variable. If expression3 is positive, the loop is ended as soon as the loop variable value exceeds that of expression2. If expression3 is negative, the loop is ended as soon as the loop variable value is smaller than that of expression2. In all other cases, the statements between FOR and NEXT are repeated with the new loop variable. In any case all statements between FOR and NEXT are executed at least once, because the test occurs at the end of the loop. If expression3 is chosen, care must be taken not to produce an endless program loop. If, for example, an 0 is given as value for the step width after STEP, then this loop has no logical end.

For example:

```
10  FOR L = 1TO10
20  PRINT L,SQR(L)
30  NEXT
```

This example prints the square roots between 1 and 10. If the loop is to run in reverse sequence (from higher values for the loop parameter to lower values) then a negative number must be given for the step width. For example:

```
10  FOR I = 100TO10 STEP -1
20  PRINT I,3.14*I
30  NEXT
```

FOR/NEXT loops may also be nested. The statements within a loop may themselves define other loops. So, the loop variable of the innermost variable runs first and the outermost loop's variable runs last.

For example:

```
10  FOR I = 0TO9
20  FOR J = 0TO9
30  PRINT 10*I+J
40  NEXT J,I
```

This small program example prints all numbers from 0 to 99 in increasing order of magnitude.

8.13 GET

Format: linenumberGETvariable

Arguments: variable is a numerical or text variable

Abbreviation: gE

The GET statement gets the next available character from the keyboard buffer and gives it to the variable. Only a single character is read. If there are several characters in the buffer, the next character can be read only by a new GET statement. If the keyboard buffer is empty, a numerical variable of 0 or a null text ("") is assigned to the variable. If a numerical variable is used, the status variable ST must also be called to find out if a 0 has been put in via the keyboard or if the keyboard buffer was empty, for in both cases the variable had a value 0. The GET statement must not be used in direct mode or the error message ?ILLEGAL DIRECT will appear. GET does not wait for a key to be pressed but always transfers a value to the relevant variable. INPUT may also be used to read data from the keyboard. GET can be differentiated from INPUT in the following ways:-

- With GET, only a single text character is read from the keyboard.

INPUT reads as many as are necessary to allocate values to all variables in the INPUT statement. INPUT must therefore wait till all variables have a value.

- GET never waits but always transfers a value to a variable immediately, even if this value is 0 or null.

GET may also be used in a program loop in order to make the program wait at that point for a valid value. For example:

```
175 GET A$:IF A$=""THEN 175:REM waits for any key.
```

8.14 GET#

Format: linenumber GET#channelnumber, variable

Arguments: channelnumber is the number of a previously OPENed data input channel.
variable is a numerical or text variable

Abbreviation: None

The GET# statement reads a single character from a device. If the device has no data prepared, then, as with GET, a numerical variable receives a 0 and a text variable receives Null (""). The data channel must previously have been opened by OPEN or DOPEN. If not, the error message ?FILE NOT OPEN will appear. If an 0 is used as device number in the OPEN statement, the GET# statement will function as GET with the keyboard. GET# also does not wait for data; if more than one text character is to be read, it is better to use INPUT#. INPUT# stops the program until all its variables have a value. GET# must also not be used in direct mode; otherwise the error message ?ILLEGAL DIRECT will appear.

When using GET#, the status variable ST should also be called to recognise the logical end of a data file (END-OF-FILE). If one tries to read from the end of a data file, GET will always transfer the character carriage return (CHR$(13)). The status variable ST receives the value 64 at the end of the file. For example:

The following example reads the contents from a floppy file character by character and prints this on to the screen. The information is read from the file in segments, each having 50 characters.

```
100  DOPEN#5, "Datafile"
110  A$=" "
115  FOR I=1 TO 50
120  GET#5, B$
130  A$ = A$ + B$
140  REM "check end of file"
150  IF B$ = CHR $(13) AND ST=64 THEN GOTO 250
160  NEXT I
170  PRINT A$
180  GOTO 110
250  PRINT A$
260  PRINT "end of file reached"
270  DCLOSE#5
280  END
```

8.15 GOSUB

Format: linenumberGOSUBlinenumber2

Argument: linenumber2 is the first line of a subprogram which should be called in by GOSUB

Abbreviation: goS

GOSUB jumps to a subprogram which begins at linenumber2. If the subprogram executes the statement RETURN, the program jumps back to the next statement after GOSUB.

A subprogram consists of a series of BASIC statements which are terminated by RETURN. Such a subprogram can be called in from various points in the BASIC program. By using GOSUB, the computer "notes" where to return on the execution of RETURN. Such a structure is useful if the same group of statements must be executed at various points of the program. They are collected at one point of the program and executed as a subprogram by using GOSUB.

```
5    A = 3
10   GOSUB 100
20   PRINT A
30   A = 10
40   GOSUB 100
50   PRINT A
60   END
100  A = A * 10
110  RETURN
```

Not only is memory space saved in this way, but also error tracking is also made easier - this is because program parts which appear at various points in the program would also have to be corrected at those points. Subprograms represent an element of structural programming.

Subprograms may be nested. If a subprogram is called in, the return jump address in noted in an internal memory area - the stack. If a subprogram is called but not left by the RETURN, the return jump address remains stored in the stack. In this case, the stack runs over and the error message ?OUT OF STACK appears. It is theoretically possible to nest a total of 23 subprograms together.

8.16 GOTO or GO TO

Format: linenumber GOTO linenumber2
or
linenumber GO TO linenumber2

Argument: linenumber2 is the linenumber of a BASIC statement in your program.

Abbreviation: gO

The GOTO statement jumps to a BASIC statement at linenumber2. It is thus possible to execute statements out of sequence. Either GOTO or GO TO may be used. If the statement in line linenumber2 is an executable statement, the program will continue with this statement. If it is not, the program will continue with the first of the executable statements after line linenumber2.

The line number must be in the GOTO statement. It is not possible to use a variable or evaluate an expression in order to determine linenumber2. For example:

This example shows how to jump to statements instead of executing them in sequence. Note that the GOTO statements jump to statement 50 after the information is printed.

```
10   INPUT "ENTER A NUMBER";A:PRINT "THE NUMBER";
20   IF A < 0 THEN GOTO 100
30   IF A = 0 THEN GOTO 200
40   PRINT A;"IS LARGER THAN";
50   PRINT "ZERO":INPUT "AGAIN? (Y/N)";Y$
60   IF Y$ = "Y" THEN 10:ELSE END
100  PRINT A;"IS SMALLER THAN";
110  GOTO 50
200  PRINT "IS EQUAL TO";
210  GOTO 50
```

8.17 IF...GOTO

Format: linenumber IF expression GOTO linenumber2

Arguments: expression is any expression (arithmetic, string or logic)
linenumber2 is the line number of a statement in your program.

Abbreviation: None

The IF...GOTO statement decides, according to the condition in expression, whether the program jumps to the statement in linenumber2. Another form of this, the IF...THEN...ELSE statement is described in Section 8.18. The total IF...GOTO statement must occupy one program line, as all BASIC statements.

Expression can contain variables, text constants, numbers and logical operators. More detailed information on the general format of BASIC expressions can be found in Chapter 5. Here are some examples of IF...GOTO statements:

```
IF A = B GOTO 500
IF (A < 50) AND (X*Y > .765) GOTO 950
IF A$ = ""GOTO150
IF LEN(S$) > 60 GOTO 1234
IF LEN(Z$) > 50 AND RIGHT$(Z$,1) = "R" GOTO 6540
```

If the conditions in expression do not comply, the statement following the IF...GOTO statement will be executed. For example:

(In this example it is decided with IF...GOTO if the SQR (square root) statement will be executed or not.)

```
100  IF X < 0 GOTO 200
110  Y = SQR(X)
120  ...further BASIC statements
200  PRINT X;"MUST NOT BE SMALLER THAN ZERO"
210  ...further BASIC statements
```

8.18 IF...THEN...ELSE

Format: linenumber IF expression THEN thenclause :ELSE elseclause

Arguments: expression is an arithmetic expression thenclause (elseclause) is a statement, a group of statements or a line number

Abbreviation: None

The IF...THEN...ELSE statement checks the condition in expression. Depending on the result, either the statement in the thenclause is executed (if expression is "true") or (if expression is "false") the statement in the elseclause is executed.

The checking in the IF...THEN...ELSE statement occurs in the following way:

1. expression is recognised as true or false. If the conditions in expression comply, then true is set and if they do not, then false is set.

2. If expression is true, the thenclause is executed (the program processing continues with this statement) and the elseclause is ignored.

3. If expression is untrue, the thenclause is jumped and the elseclause executed.

The line is processed from left to right in an IF...THEN...ELSE execution. All statements following THEN and finishing either with ELSE or at the end of the line, are regarded as the thenclause. All statements which follow ELSE and finish with the end of a line are regarded as the elseclause. Without ELSE, the program processing will continue in the next line if expression is untrue. ELSE and the elseclause must be in the same line as the relevant IF...THEN statement. ELSE and elseclause cannot be used without the IF..THEN statement. The thenclause or the elseclause could look like this:

- Single BASIC statements:-

```
A = B

NAME$ (I) = INNAME$

X = SQR(Y*Z) + ATN(NEW VALUE)

INPUT"ENTER THE CORRECT VALUE";VALUE
```

- or a group of BASIC statements:-

```
A = B: X = R*3

N% = N% + 1:NAME$ (I) = INNAME$

R = .5: A*B*C: GOTO 500
```

- or the line number of a BASIC statement in your program.

If an IF...THEN...ELSE statement is used, a colon must be placed in front of ELSE. For example:

```
100  IF A = 150 THEN B = A:ELSE B = 0
```

It is also possible to omit ELSE if it is not required. For example:

```
100  IF A = B THEN A = .5*B
```

Further, THEN can be made ineffectual by placing only a colon after THEN. If expression is true, no thenclause will be executed and processing will continue in the next program line. If expression is untrue, the elseclause will be executed. For example:

```
100  IF A = B THEN: ELSE A = (B/.5)
```

The IF...THEN...ELSE statements may be nested within other IF...THEN...ELSE statements in an elseclause. The IF...THEN...ELSE without the ELSE can also be a thenclause. Examples of nested IF...THEN...ELSE statements are:

```
IF A=B THEN X=0:IF A<B THEN X=-1:ELSE X=50

IF LEN(N$)=0 THEN 500:ELSE IF LEN(A$)>30 THEN N$=A$

IF X=Y THEN X=Y/2:ELSE IF R<.99 THEN X=R:ELSE Y=R/5
```

The entire IF...THEN...ELSE statement, including the nested one, must fit into the one program line, like all BASIC statements. A line number may be in the thenclause or the elseclause. If this is the case, the program jumps to the line with this line number and continues processing at this point. For example:

```
IF X<0THEN 30: ELSE 500

IF NAME$ = ""THEN 650: ELSE NAME$ = NAME$ + ADD$

IF I%<95 THEN NAME$(J) = A$: ELSE 780
```

The IF...THEN...ELSE statement may also be used in direct mode. Care must be taken that a given line number is available as jump address. A line with this line number must previously be given, together with a BASIC statement. If such lines are absent,

?UNDEFINED STATEMENT will appear.

If an IF statement is given in direct mode and causes a jump to a program line, the processing continues in program mode from this line on.

If a test on equality is executed in expression and the variables are stored in real form, care must be taken because the computer may not store an exact value. A small variation margin should therefore be left. For example:

If one needs to check whether a real variable A is equal to 0.1, a variation margin of 0.000001 is left so that the statement reads:

```
IF ABS(A-0.1)<=1.0E-6 THEN...:ELSE...
```

This test on equality of real variables ensures that the real equality is tested with a defined deviation. The same sort of test can cause problems if a STEP variable of non-INTEGER type is being processed in a FOR statement. For example:

1) In this example it is shown how the square root of a positive number is printed:

```
100  N$ = "THE VALUE MUST BE POSITIVE: REENTER"
110  P$ = "THE ROOT IS"
120  INPUT"ENTER A NUMBER";N
130  IF N<0 THEN PRINT N$: GOTO 120: ELSE PRINT P$:SQR(N)
140  INPUT "ANOTHER NUMBER(Y/N)";Y$
150  IF Y$ = "Y" THEN 120:ELSE END
```

2) Here it is seen how a value is tested to determine whether it is in the correct range:

```
100  IF(I<50)OR(I>100) THEN 500: ELSE R=I:X=I/2
110  REM VALUE IN CORRECT RANGE
120  . . . .
500  REM VALUE OUTSIDE THE RANGE
510  . . . .
```

8.19 INPUT

Format: linenumber INPUT prompttext; variablelist

Arguments: prompttext is a text which is enclosed in inverted commas(") ~~or a text variable~~
variablelist is a list separated by commas of one or more variables.

Defaults: prompttext="", ie. Null.

Abbreviation: None

The INPUT statement first writes prompttext with a question mark at the end and then reads the values from the screen into the variablelist. The program waits till enough values for the entire variablelist have been given. INPUT statements enable information from the user to be given via the screen to the program. INPUT takes the first symbol as the start of a value. Values end with carriage return or a comma. The INPUT statement functions in the following manner:

1. Prompttext is written with question mark on the screen. If there is no prompttext, only the question mark is printed.
2. The values are given to the screen and read into the variablelist.
3. If more data are needed, 2 question marks appear on the screen and the program waits until more data is entered.
4. The values are given in the order in which they appear in the variable list.
5. If the RETURN key is pressed without input, the variable keeps the value it had previously.

The variable names in the variable list can be any BASIC names, including integer, real, text and array variables. The given type of value must correspond with the type of variable in the variable list. If the attempt is made to use INPUT in direct mode, the error message ?ILLEGAL DIRECT appears. Commas are used in the INPUT statement to separate values from each other if the variablelist has more than one variable. When text variables are given, inverted commas should only be used if the text to be input contains commas, colons or semicolons. As many as 158 symbols (corresponding to one logical line less space for the prompt) may be entered. If more than one logical line is to be entered, carriage return must be operated to indicate the end of the first part of data. Then the computer "knows" that more is to follow and 2 question marks immediately appear on the screen. Data input can continue straight away. The 2 question marks stay on the screen until all variables in the variablelist have received a value. Care must be taken that integer variables do not have a figure after the decimal point. If a number with a figure after the decimal point is entered, it is simply ignored. Using INPUT:

```
INPUT "ENTER I,J";I%,J%
```

and entering the values:

```
1.23,45.6789
```

the variables will assume the following values:

```
I%=1, J%=45
```

The INPUT statement only transfers the entered value to the corresponding variable if the two types correspond. The following errors can occur:

- If values of the wrong type are entered (i.e. text characters for numerical variables) the error message ?REDO FROM START will appear.

- If too many values are entered (i.e. more than on the variablelist) the excess values are ignored and the message ?EXTRA IGNORED appears.

Here it can be seen how an INPUT statement can be used without prompttext:

```
10  INPUT I%,J%
20  PRINT I%,J%
RUN
? 123,456
123      456
READY
```

A further example of INPUT:

```
10  FOR I = 1 TO 10
20  INPUT "ENTER NAME AND HOURS"; NA$(I),H(I)
30  T = T + H(I)
40  NEXT I
50  PRINT "NAME","HOURS"
60  FOR I = 1 TO 10
70  PRINT NA$(I),H(I)
80  NEXT I
90  PRINT "TOTAL HOURS = ";T:END
```

8.20 INPUT#

Format: linenumber INPUT# channelnumber, variablelist

Arguments: channelnumber is the logical number of the file which is to be read. Channelnumber can be any number between 1 and 255
variablelist is a list of variables, as in the preceding section (8.19).

Abbreviation: iN

The INPUT# statement reads values from the logical file channelnumber and uses them as the variables in variablelist. The INPUT# functions just like INPUT with the difference that the values are read from a file and not from the screen. The file must be opened with OPEN (see 8.25) before using INPUT#. The values to be read from the file must be in the same sequence as the variables in the variablelist and are allocated to the variables correspondingly. It must be ensured that the correct variable type for the relevant variable is received. Leading spaces are ignored by INPUT# if data are read from the file. Numbers and texts must end with carriage return, line feed or a comma. The INPUT# statement only allocates the entered value to the variable when they are of the same type. If, for example, a numerical variable receives a text value, the error message ?FILE DATA ERROR appears. For example:

Here it can be seen how a file is opened to a disk drive and how data are read in with INPUT#:

```
 5  A=1:B=2:C=3
10  OPEN 1,8,2, "MY DISK FILE"
20  PRINT "THE DISK FILE IS OPEN"
30  INPUT# 1,A,B,C
40  CLOSE1 : PRINT A,B,C
50  END
```

8.21 LET

Format: [linenumber] [LET] variable = expression

Arguments: variable is any BASIC variable name
expression is a BASIC statement of the same type

Abbreviation: [lE]

The LET statement allocates the value of expression to the variable. LET is an allocation statement or value allocator. LET is not obligatory and is normally omitted. LET A=B is the same as A=B.

LET can be used with any numerical, text or array variable, or internal or self-defining function. For example:

```
LET B=1             Sets B equal to 1

LET X=SQR(Y*Z/2)    Is the same as X=SQR(Y*Z/2)
```

8.22 NEXT

Format: linenumber NEXT [variable [,...,variable]]

Arguments: variable is the variable which was determined in the relevant FOR statement

Abbreviation: nE

The NEXT statement is at the end of a FOR loop. (More details on FOR loops can be found in 8.12.)

Example of a FOR loop:

```
100  FOR I = 1 TO 3
     ...BASIC statements
200  NEXTI
```

The NEXT statement in line 200 closes the FOR loop which began in line 100.

If NEXT is used without the variable parameter, NEXT will affect the FOR loops which immediately preceded it. If FOR loops are nested then:

```
100  FOR I = 1 TO 10
110  FOR J = 34 TO 50
     ...BASIC statements
200  NEXT
210  NEXT
```

The NEXT statement in line 200 affects the FOR loop which begins in line 110 (i.e. the one immediately preceding) and the NEXT statement in line 210 affects the FOR loop beginning in line 100.

If the parameter variable is given in NEXT when using nested FOR loops and the FOR loop in question is not the one immediately preceding, then the FOR loops are processed erroneously. The NEXT statement then works on the FOR loop with the parameter mentioned. The FOR loop immediately preceding with a different step parameter is aborted.

Several parameter variables can be determined if it is necessary to terminate several FOR loops in the same line. The above example could also appear:

```
100  FOR I = 1 TO 10
110  FOR J = 34 TO 50
     ... BASIC statements ...
200  NEXT J,I
```

This NEXT in line 200 first closes the FOR loop with parameter J and then with parameter I. Up to 10 FOR loops may be terminated by a single NEXT statement. If NEXT is used without the relevant FOR statement, the error message

?NEXT WITHOUT FOR appears.

Care must be taken when nesting FOR loops that the NEXT statement corresponds to the correct (the one immediately preceding) FOR loop. If the NEXT statement is omitted, all BASIC statements are executed to the end of the program. For example:

1) Here, several numbers are printed using FOR loops. There are two NEXT statements, one for each loop:

```
100  FOR I = 1 TO 2
110  FOR J = 2 TO 3
120  PRINT "I" I "J" J
130  NEXT J
140  NEXT I
RUN

I  1  J  2
I  1  J  3
I  2  J  2
I  2  J  3
READY
```

2) The same FOR loops are used but one NEXT statement with two parameters is used to close:

```
100  FOR I = 1 TO 2
110  FOR J = 2 TO 3
120  PRINT "I" I "J" J
130  NEXT J,I
```

3) It can be seen here which errors are produced by this program if the NEXT statement refers to the false FOR loop:

```
100  FOR I = 1 TO 2
110  FOR J = 2 TO 3
120  PRINT "I" I "J" J
130  NEXT I
RUN

I   1   J   2
I   2   J   2
READY
```

8.23 ON...GOSUB

Format: linenumber ON expression GOSUB linelist

Arguments: expression is an arithmetic expression
linelist is a list of line numbers of one or more subprograms. The line numbers must be separated by commas.

Abbreviation: None

The ON...GOSUB statement tests the value in expression and calls in one of the subprograms whose line numbers are in the linelist. The jump to subprogram with GOSUB is described in Chapter 8.15.

This is how the ON...GOSUB statement functions:-

1. Expression is checked first. If the value is not integer, it is treated as one by ignoring the figures after the comma.

2. After this, there is a jump to a subprogram from linelist. If expression is equal to 1 the jump will be to the first line number in the linelist. If the expression is equal to 2, to the second line number, etc.

3. If expression is 0 or longer than the number of line numbers in the linelist, the statement following the ON...GOSUB statement will be executed. In this case, no subprogram is processed.

4. After processing the subprogram, the statement following the ON...GOSUB will be executed.

Each line number in the linelist must be one in the program which initiates a subprogram. Otherwise the error message ?UNDEFINED STATEMENT appears. The value in expression must be larger than or equal to 0. If expression is a negative value, the error message ?ILLEGAL QUANTITY appears. The ON...GOSUB statement is a very important aid to the structured construction of many programs.

8.24 ON...GOTO

Format: line number ON expression GOTO linelist

Arguments: expression is an arithmetic expression.
linelist is a list of line numbers of statements in the program. The line numbers must be separated by commas.

Abbreviation: None

The ON...GOTO statement checks the value in expression and jumps to one of the line numbers from linelist. More information on line jumps is to be found in Section 8.16, in connection with the GOTO statement.

ON...GOTO functions in the following way:

1. expression is checked first. If the value is not an integer it will be treated as one by ignoring the figures after the comma.

2. After checking the value in expression, a jump is made to a statement with a line number from linelist. If expression is equal to 1, the jump will be to the first line number in the list and if expression is equal to 2, to the second line number, etc.

3. If expression is equal to Ø or larger than the number of line numbers in linelist, the statement following ON...GOTO will be executed. In this case, no jump occurs.

Every line number in linelist must be a line number found in the program. Otherwise the error message ?UNDEFINED STATEMENT appears.

The value in expression must be larger than or equal to Ø. If it is a negative value, the error message ?ILLEGAL QUANTITY appears.

Ensure that an integer variable is allocated to the value in expression. If another value is given, the figures after the decimal point are simply ignored. (i.e. If value 2.345 is entered, the computer stores value 2 and the 2nd line in the linelist is used.) For example:

```
10   INPUT "ENTER A NUMBER";X
20   IF X<0 THEN GOTO 500
30   ON X GOTO 100,200,300
40   PRINT "YOUR NUMBER WAS ZERO OR LARGER THAN THREE"
50   INPUT "AGAIN?(Y/N)";Y$
60   IF Y$ = "Y" THEN GOTO 10:ELSE STOP
100  PRINT "YOUR NUMBER WAS EQUAL TO ONE"
110  GOTO 50
200  PRINT "YOUR NUMBER WAS EQUAL TO TWO"
210  GOTO 50
300  PRINT "YOUR NUMBER WAS EQUAL TO THREE"
310  GOTO 50
500  PRINT "YOUR NUMBER WAS NEGATIVE"
510  GOTO 50
600  END
```

8.25 OPEN

Format: linenumber OPEN channelnumber, devicenumber [,secondaryaddress],[filename]

Arguments: channelnumber is the logical number which is allocated to the file. It can be any number between 1 and 255.
devicenumber is the number of the device. It may be any number between 0 and 255, depending on the devices connected. (Normally only 0 to 15 are valid.)
secondaryaddress is a number which is sent to the device.
filename is the name of the file and may include special characters.

Abbreviation: oP

The OPEN statement, coordinates a I/O channel to an external device such as a disk drive or printer. The OPEN statement must be used to achieve a connection between a file and a device and between a device and channel number before using a GET#, INPUT#, or PRINT# statement on a device or file.

The channelnumber is also called logical file number and must always be given in the GET#, INPUT# and PRINT# statements. If, for example, a file is to be opened to the printer with channel number 6, then all corresponding PRINT# statements must be written as PRINT#6... Devicenumbers are primary addresses of systems to which special devices are allocated.

The secondaryaddress parameter can be determined according to the following Table:

OPEN Commands : secondary addresses

Device	Secondaryaddress	Effect
Disk	1-14	Opens a data channel
	15	Opens a command channel
Keyboard	1-255	None
Screen	1-255	None
Printer	1-255	See Printer handbooks
RS232	1 or 129	Opens an output channel
	2 or 130	Opens an input channel
	3 or 131	Opens a bidirectional channel

The filename parameter is sent to the device upon opening. The value given to this parameter depends on the device in question. If a disk file is opened with the parameter secondaryaddress = 15, control information can be transferred with filename. The RS232 interface is described in more detail in another section.

The various forms of OPEN statement must have been understood before effectively using them with the GET#, INPUT# and PRINT# statements.

Examples:

OPEN 1,0	Opens the keyboard as channel 1
OPEN 6,4,0	Opens a logical channel 6 to the printer
OPEN 7,4,7	Opens another channel to the printer.
OPEN 11,8,1,"DISKDATA,S,W"	Opens logical channel 11 to disk drive (device 8) to write a sequential file called "DISKDATA".

8.26 POKE

Format: (linenumber) POKE address, value

Arguments: address is a memory location. This is an integer between 0 and 65535 (i.e. 16 bit)
value is an integer between 0 and 255. (i.e. 8 bit.)

Abbreviation: pO

The POKE statement writes the value into the memory address in the memory bank last selected by a BANK statement.

POKE does not check if the given address exists in the available RAM, but puts the value on the bus and sends it to the address. If the address is smaller than 0 or larger than 65535, the error message ?ILLEGAL QUANTITY appears.

Addresses and values must be integers. If a real variable is used, the figures after the decimal point are ignored. For example:

```
POKE 12345,23.56
```

The value 23.56 is ammended so that the statement actually becomes:

```
POKE 12345,23.
```

If text variables are entered for address or value, the error message

?TYPE MISMATCH appears.

As each memory cell is only capable of taking one single memory word byte, the value of a number must be between 0 and 255. If the value is smaller than 0 or larger than 255, the error message

?ILLEGAL QUANTITY is given.

The built-in PEEK function is often used with POKE to store data, to reach assembler subprograms in the working memory, to give the assembler information and to obtain results from the assembler subprogram. You will find more information on this in later sections.

8.27 PRINT

Format: (linenumber) PRINT printlist

Arguments: printlist is text, variable names, expressions or functions.

Defaults: printlist = blank text, a line feed will occur.

Abbreviation: ? (question mark)

The PRINT statement writes the printlist on the screen. The question mark can be used instead of PRINT when entering BASIC statements. If the program is then printed, PRINT appears for the question mark in the list. For example:

```
PRINT A,B
PRINT "THE ANSWER IS" A$
PRINT EXP(Y*Z)+Y
PRINT SUBTTL% "THE VALUE IS ZERO"
PRINT A;B;
PRINT A,B
```

Strings in the printlist must be enclosed in inverted commas. The PRINT statement decides where the values are to be printed on the screen depending on the punctuation. BASIC divides each print line into segments which can contain 1Ø characters. Tabulator stops are used at every tenth position. Punctuation in the printlist has the following influence on the PRINT statement:

- If two expressions on the printlist are separated by commas, the 2nd expression is printed at the following tabulator stop, i.e. in the following segment.

- If 2 expressions are separated by a semicolon, the 2nd expression is printed directly after the first.

- One or more spaces between two expressions have the same effect as a semicolon.

- If there is a comma or semicolon after the last expression on the printlist, the next PRINT statement prints its printlist after the first. The distances are determined by punctuation symbols. With no comma or semicolon at the end, the next PRINT statement starts a new line.

If the print line is longer than a screen line, PRINT will write the remaining values in the next screen line.

The expressions are printed as follows:

- One position is always jumped after numbers.
- A space is always in front of a positive number and a minus sign before a negative number.
- Numbers with more than 1Ø places and numbers between Ø and Ø.Ø1 are always printed in exponential notation.

The Series 700 computers have an enlarged PRINT USING statement with which formatted lines can be printed. Special print formats are then possible.

The PRINT statement can print many special characters in addition to the text characters and numbers. The following section shows how to enter these special characters.

Quotes mode:

After using the quotes key (") the computer is in quotes mode. Number and letter keys are unchanged, but all other keys, such as the cursor, write their ASCII character in the printlist instead of executing the given cursor function directly. Different control information can be written into the print list in this way.

To leave the quotes mode, the escape key must be used (ESC), or " again. All keys then revert to normal use again.

The DEL key is not affected by the quotes mode. The following control information may be transferred in the quotes mode:

- Cursor movement and other special characters
- Reverse characters.

The INS key can also be used to produce spaces in the printlist.

Cursor control in quotes mode

Every cursor movement key can be used in quotes mode. The control possibilities are listed individually in the appendices.

Output of inverted characters (reverse)

Inverted characters appear on the screen as dark on light background instead of light on dark. Inverse characters are entered in quotes mode after pressing the RVS key. Firstly an inverted r (for reverse) appears which indicates the start of the inverted characters. This letter is not printed during execution of the PRINT statement but serves only as a marker. Any character may now be entered. They will appear on output as inverted characters. If the text with inverted characters is finished, pressing the key OFF will return it to normal. At the end of this text there will be an inverted R as marker. The return key can also be used to end the printing of inverted characters. After a PRINT statement with inverted characters, the computer automatically returns to normal presentation. If, however, there is a comma or semicolon at the end of the statement, the inverse presentation is maintained and the characters of the next PRINT (which will be printed in the same line) will also appear in inverse form. For example:

To obtain HALLO in reverse form, enter:

```
PRINT "RVS HALLO OFF"
```

8.28 PRINT#

Format: [linenumber] PRINT# channelnumber, printlist

Arguments: channelnumber is the logical number of the file which was priviously opened by OPEN or DOPEN. printlist is a text, variable names, expression, or function.

Abbreviation: pR (Attention: not ?#)

The PRINT# statement writes the printlist in the file defined by channelnumber. If the file referred to by channelnumber has not previously been correctly opened, the error message ?FILE NOT OPEN appears.

The PRINT# statement functions just like PRINT, with the difference that in this case a file with the relevant channelnumber is used. The data are transferred in the same manner as in the PRINT statement:

- As for PRINT, values separated by commas are divided into segments which are 10 characters long (padded with spaces).

- Values separated by a semicolon or spaces are printed consecutively.

- A carriage return is automatically written as the last character of the file line if no comma or semicolon is on the printlist as last character.

INPUT statements read from file data which have been written with PRINT#. Text variables should always be within inverted commas and numbers separated by commas. For example:

```
10   OPEN 1,8,1, "MY DISKFILE,S,W"
30   C$ = CHR$(44)
40   ...some BASIC statements
200  PRINT# 1,A,C$,B,C$,D
210  PRINT# 1, "NAME"
220  PRINT# 1,1,C$,2,C$,3.
230  END
```

8.29 PRINT USING and PRINT# USING

Format: [linenumber]PRINT[#channelnumber,] USING formatlist printlist;

Arguments: channelnumber is the logical number of a file previously opened by OPEN
formatlist defines the format of the expressions in printlist.
printlist is a list of expressions to be printed separately by commas.

Abbreviations: ? or pR & usI

A formatlist can be defined with PRINT USING which determines the form of the data in the printlist. PRINT USING uses the screen and PRINT# USING uses a file, in the same manner as PRINT and PRINT#. The PRINT(#) USING statement is in principle a PRINT(#) statement with explicitly defined data formatting. PRINT(#) however, writes the data in standard format (as described earlier).

These are the main differences between PRINT(#) and PRINT(#) USING:-

- TAB and SPC functions cannot be used in the print list of PRINT(#) USING
- Semicolons between expressions in the printlist cannot be used in PRINT(#) USING
- Semicolons may only be used as termination of the printlist as for PRINT(#)
- The expressions from printlist of the PRINT(#) USING statement are separated by commas. They have no influence, however, on the format, as in PRINT(#).

The USING clause consists of USING and the Formatlist. The Formatlist consists of one or more 'format arrays'.

A 'format array' has format characters from the following table. If characters other than these are used, they will appear in the print itself; they have no formatting function. The legibility of the output is thus increased. An expression from the printlist is described with every format array. If there are more expressions in the printlist than the formatlist, the formatlist is re-used as often as necessary.

Formatting characters

Character	Meaning
Hash sign(#)	Each hash sign in a format array reserves space for one character. Each format array must have at least one hash sign.
Plus(+) and minus(-)	Plus and minus can either be the first or last position of the format array. The operational sign of the number is printed at the given point.
Decimal point(.)	The decimal point of a number is determined by THIS. Only one decimal point per format array. THIS SIGN can be altered with a PUDEF statement.
Comma(,)	With a comma in a number, longer numbers are more easily read. This character can be altered with a PUDEF statement.
Dollar sign($)	A $ is printed in front of the first valid digit of a number. This character may be altered with a PUDEF.
Four arrows (↑↑↑↑)	If a format array ends with or contains four arrows which are in turn followed by a plus or minus sign, the number is printed in Exponential format.
Equals sign(=)	Texts are normally printed on the left. They are centred by using the equals character.
Larger than(>)	Using this sign, the texts will appear on the right.

The characters in the format array belong to number or text variables as can be seen from the following table. Text format symbols in format arrays can be taken for number expressions and vice versa. If format symbols are mixed, however, they will be interpreted as hash signs(#) and will lose their special formatting function.

Format character types

Character	Number formatting	Text formatting
Hash sign (#)	X	X
Plus (+)	X	
Minus (-)	X	
Decimal point (.)	X	
Comma (,)	X	
Dollar ($)	X	
Four arrows (↑↑↑↑)	X	
Equals sign (=)		X
Larger than (>)		X

The hash sign is used for text and numerical variables. If also reserves space for a character in the output array. If the data expression has more space than prepared by the #, then the following occurs:

- # In a number variable: The entire array is filled with asterisks (*) and no number is printed.

- # In a text expression: All prepared spaces are occupied, excess data is ignored.

If an array is to be produced which has a maximum of 7 characters, the following PRINT USING instruction is entered:

```
PRINT USING "#######"; NAME$
```

If NAME$ has more than 7 characters, the eighth and all subsequent characters will be ignored.

To print a number with a maximum of 4 places, we use:-

PRINT USING "####";A With this formatting statement the program prints:

```
A=12.34        12
A=567.89      568
A=123456     ****
```

The plus and minus characters can either be printed first or last in the format array. The plus prints a plus sign and the minus a minus sign.

If a minus sign is entered and the number is positive, a space is printed.

If more text variables are available than are defined in the format array, then the characters appearing on the right which are superfluous are simply ignored.

Examples:

Array	Expression	Result	Comment
+##	1	+b̸1	Blank between operational sign and number
#.##+	-.01	0.01-	Leading 0 added
-.##	-.1	-.10	Leading 0 suppressed by minus sign
##.#-	1	1.0	Trailing 0 added
+###+	1	ERROR	Two plus signs
+##.#-	1	ERROR	Plus and minus signs
####	-100.5	-101	Rounded to a total 4 characters
####	-1000	****	Overflow, as 5 characters do not fit into array
#.##	-4E-03	-.00	Rounded to a total 4 characters
##.	10	10.	Decimal point added
#.#.	1	ERROR	Two decimal points
##,##	-10	-10	Minus has priority over comma
##=<<	1000	1000.0	= and < are treated as #, since they are in number array
#$##	1	$1	Preceding $ character
+#.#↑↑↑↑	1	+1.0E+00	Expression in exponential format
#↑↑↑↑+	-340	3E+02-	Trailing sign
##↑↑↑	1	ERROR	Only three arrows
####	cbm	cbmb̸	Text expression printed on the left
###>#	cbm	b̸b̸cbm	Text expression printed on the right in a 5-character array
#####	cbm	cbmb̸b̸	On the left in a 5-character array
=####	cbm	b̸cbmb̸	Centred in a 5-character array
#,$#=+	cbm	b̸b̸cbmb̸	Only the = has a control effect The other characters are treated as #

8.30 PUDEF

Format: linenumber PUDEF controlstring

Arguments: controlstring consists of 1-4 characters which are enclosed in inverted commas, or a text variable which contains 1-4 characters.

Abbreviation: pU

The PUDEF defines the symbols of a PRINT USING statement so that, for example, instead of a space, a question mark is printed. Each of the positions in the controlstring represents a certain symbol from the PRINT USING statement which can be altered.

The positions correspond to the following symbols: "␢, . $"

- Position 1 is the fill character. Default is space.
- Position 2 is the comma, with comma default.
- Position 3 is the decimal point, with the decimal point as default.
- The currency character is at position 4. Default is $.

PUDEF only alters the character if a PRINT USING is used for output.

PRINT outputs are not influenced by PUDEF.

The format array of PRINT USING is not changed at all. The symbols in the format array are not changed if the PUDEF statement is used.

To change the symbols with the PUDEF statement, the required characters must be used in the corresponding positions of the controlstring. If the space should be replaced by a question mark, for instance, then this PUDEF statement should be entered:

```
PUDEF "?"
```

Now every space will be replaced by a question mark at printout. So the expression:

```
"   12.3"
```

is printed as "???12.3"

If fewer than four characters are in the controlstring, the remaining symbols receive their default values. If more than four characters have been entered, the superfluous symbols are ignored. For example:

1) The comma and decimal point characters of a PRINT USING statement are to be exchanged:

```
10  PUDEF " .,"
20  PRINT USING "###,###,###.##";-1234.567
RUN

-1.234,57
READY
```

2) Asterisks (*) are to be printed for every space. In this example, two possibilities are offered.

```
10  PUDEF "*"
20  DATA 1.50, 2583.1, 3456789.55, .25
30  F1$ = "$##,###,###.##" :REM LEADING SIGN
40  F2$ = "#$#,###,###.##" :REM FLOATING SIGN
50  FOR I = 1 TO 4
60  READ A
70  PRINT USING F1$ ; A
80  NEXT I
90  RESTORE
100 FOR I = 1 TO 4
110 READ A
120 PRINT USING F2$ ;A
130 NEXT I
**********1.50
$*****2,583.10
$*3,456,789.55
$*********0.25
*********$1.50
*****$2,583.10
*$3,456,789.55
*********$0.25
READY
```

8.31 READ

Format: (linenumber) READ variablelist

Arguments: variablelist is a list of variable names, separated by commas.

Abbreviation: rE

The READ statement refers to one or more DATA statements and these data are allocated to the variables in the variablelist.

READ and DATA statements are often used to obtain initial values in a program.

Variablelist can contain any numerical, text or array variable names.

A READ statement can receive values from several DATA statements and different READ statements can use the same DATA statement. The data are read from the DATA statement in sequence and allocated to the variables on the variablelist. A READ statement does not have to read all values from the DATA statement. If it is not done, the next READ statement continues the processing of the DATA statement at the point where the first stopped. If more values are to be read than are in the DATA statements, the error message ?OUT OF DATA appears. If there are more data in the DATA statement than are read by the READ statement, the extra data are ignored. If the value allocated to a variable in this manner does not correspond to the variable type, the error message ?SYNTAX ERROR appears (referring to the dataline).

DATA statements as all BASIC statements, have a line number. Using RESTORE, data from a DATA statement can be reused. For example:

1) Here it can be seen how values can be read from different DATA statments by using READ:

```
10  DATA 1.0,2.5,3.8,4.9,9.9
20  DATA 11.0,12.5,14.8
30  REM READ THE INITIAL VALUE
40  FOR I = 1 TO 4
50  READ INIT(I)
60  NEXT I
70  READ PERCENT,IY,X
80  ...Rest of BASIC program
```

2) Here numerical and text variables are read with READ:

```
10  DATA 1.1,2.2,3, "TEXT ONE", "TEXT TWO"
20  DATA 4.4,"        TEXT THREE        ",5
30  READ X,Y,Z%,A$
40  PRINT X,Y,Z%,A$
50  READ B$,XYZ
60  PRINT B$,XYZ
70  READ C$.N%
80  PRINT C$,N%
90  END
RUN
1.1       2.2      3  TEXT ONE
TEXT TWO            4.4
TEXT THREE          5
READY
```

8.32 REM

Format: (linenumber) REM text

Arguments: text is any remark.

Abbreviation: None

The REM statement is a non-executable statement in the program. Any letters or characters can be in text. REM statements are regarded as the last statement of the line and may also contain colons which would otherwise mark the boundary of a statement.

REM statements are often used to write explanations into the program so that the program is easier to understand, or to explain the meaning of the variables. A possible correction to the program is made easier also.

REM statements can also be the only statement on a line, for example:

```
1ØØ  REM THIS PROGRAM WAS WRITTEN ON 7.9.84
```

When using capitals or graphic characters, the text must be enclosed within inverted commas.

The line numbers of REM statements can be jump addresses of a GOTO or GOSUB statement, but this is considered "bad programming". Examples:

Many examples of the REM statement can be found in this handbook.

Some typical ones are:-

```
1Ø   REM THIS PROGRAM WAS WRITTEN BY F.D.
25   REM THIS DATA STATEMENT CONTAINS INITIAL VALUES
3Ø   REM FOR THE AREA IN QUESTION
1ØØ  X = SQR (Z*T): REM CALCULATION OF THE SURFACE
```

8.33 RESTORE

Format: (linenumber) RESTORE [linenumber2]

Arguments: linenumber2 is the line number in the program

Defaults: linenumber2 is the line number of the first DATA statement in the program

Abbreviation: reS

By using the RESTORE statement, the following READ statement reads the value of the DATA statement in linenumber2. In Section 8.7 you will find more information on DATA statements and in 8.31 on READ statements.

If a linenumber2 is given which is not in the program, the error message ?UNDEFINED STATEMENT appears.

Linenumber2 need not be the line number of a DATA statement in the program. In this case BASIC seeks the next DATA statement after linenumber2.

READ statements normally read the values of DATA statements in sequence. By using RESTORE, however, it is possible to let data be read twice because the following READ statement begins with the DATA statement which is in linenumber2. For example:

1) 10 RESTORE The first DATA statement of the program is read.

2) 100 RESTORE 50 Then the DATA statement in line 50 (or the one following line 50) is read.

3) In the following example, the DATA statement in line 20 is re-read:-

```
10   DATA 1,2,3,4
20   DATA 5,6,7,8
30   FOR L = 1 TO 8
40   READ A:PRINT A
50   NEXT L
60   RESTORE 20
70   FOR I = 1 TO 4
80   READ A:PRINT A
90   NEXT I
100  END
RUN
1
2
3
4
5
6
7
8
5
6
7
8
READY
```

8.34 RESUME

Format: linenumber RESUME [NEXT(linenumber2)]

Arguments: linenumber2 is the line number of a BASIC program statement

Defaults: linenumber2 is the line number which caused the error

Abbreviation: resU

The RESUME statement functions in error trapping by continuing processing the program after the error has been found and processed with a subprogram. The TRAP statement described in 8.38 traps the errors.

If RESUME is used without NEXT or linenumber2, the program processing recommences at the statement where the error occurred. If the error occurs in a line with several statements, only the statement with the error will be repeated.

If the NEXT parameter is given in the RESUME statement, the processing will continue with the statement which follows the error. If there are more statements on one line, the processing will continue with the next statement in the same line.

If linenumber2 is given, the program processing will continue on that line.

The RESUME statement may not be used in direct mode. If this is done, then the error message ?ILLEGAL DIRECT will appear. Error trapping will stop when an error has occurred. RESUME switches the error trapping on again and uses the error parameters ER (error number) and EL (error line).

If you try to use a RESUME statement without the preceding TRAP statement, the error message ?UNABLE TO RESUME appears. For example:-

It can be seen here how an error is found and how to use RESUME, depending on the type of error. If there is an OUT OF DATA error (ER = 23) after line 500, the data in DATA statement 85 should be RESTOREd. In every other error, the program should be stopped.

```
10   REM IF THERE IS AN OUT OF DATA ERROR AFTER LINE 500
20   REM THE DATA MUST BE RESTORED
30   REM WITH A RESTORE 85
40   ... BASIC statements
80   DATA ...
85   DATA ...
90   DATA ...
100  TRAP 900
110  ... BASIC statements
600  READ A,B,C,D,E: REM HERE IS AN OUT OF DATA
610  ... BASIC statements
900  REM START OF ERROR TREATMENT
910  REM ONLY THE OUT OF DATA ERROR (ER = 23) AFTER LINE 500
920  REM SHOULD BE TREATED.  IN EVERY OTHER ERROR
930  REM THE PROGRAM PROCESSING SHOULD BE STOPPED
940  IF (ER <> 23) OR (EL < 500) THEN STOP
950  REM THERE IS AN OUT OF DATA AFTER LINE 500
955  REM ERROR OCCURRED
960  RESTORE 85
970  RESUME
980  END
```

8.35 RETURN

Format: (linenumber) RETURN

Abbreviation: reT

The RETURN statement is the last statement of a subprogram and activates the jump to the statement following the GOSUB call. More detail on the GOSUB is to be found in Section 8.15.

Sub-program statements can be anywhere in the BASIC program. If the subprogram is placed at the end of a program, the final END can be put in front of the start of the subprogram so that the subprogram cannot in any circumstances be executed without the GOSUB statement. If a program finds a RETURN without a preceding GOSUB, there is the error message ?RETURN WITHOUT GOSUB. For example:-

```
10   PRINT "PROGRAM START"
20   PRINT "CALL UP FIRST SUBPROGRAM"
30   GOSUB 200
40   PRINT "CALL UP SECOND SUBPROGRAM"
50   GOSUB 300
60   PRINT "COMPLETED"
70   END
200  REM THIS IS THE FIRST SUBPROGRAM
210  PRINT "IN THE FIRST SUBPROGRAM"
220  RETURN
300  REM THIS IS THE SECOND SUBPROGRAM
310  REM THIS SUBPROGRAM CALLS A THIRD SUBPROGRAM
320  PRINT "IN THE SECOND SUBPROGRAM; A THIRD IS CALLED"
330  GOSUB 400
340  RETURN
400  REM THIS IS THE THIRD SUBPROGRAM
410  PRINT "IN THE THIRD SUBPROGRAM"
420  RETURN
```

8.36 STOP

Format: (linenumber) STOP

Abbreviation: sT (not to be confused with the reserved word ST)

The STOP statement ends program processing and returns to direct mode. The STOP statement does not close files. Processing can continue with CONT after having been stopped by STOP.

STOP statements can be anywhere in the program. The program is purposely interrupted and statements can be given in direct mode in order to change or examine variables, for example. Processing can resume with CONT. For example:-

```
20    INPUT "ENTER A NUMBER";X
30    Y = SQR(ABS(X))
40    Y = Y*X
50    X = X/100
60    IF (Y < 1.0E-040) OR (Y > 1.0E+20) THEN STOP
70    PRINT "THE ACTUAL VALUES ARE",X,Y:END
```

8.37 SYS

Format: (linenumber) SYS address

Arguments: address is the address of a machine code program. It can be either a variable or the address itself.

Abbreviation: sY

SYS statements permit BASIC to be mixed with machine code in a single program.

SYS statements can be used in direct and program mode. They are often used to call up subroutines of the operating system in Bank 15.

Address is the address of the start of the machine code program in the memory. Address can be:

1) The name of a variable which has this value, for example:

```
100  MYSUB = 30200
     ...
120  SYS MYSUB
```

2) The address of the machine code program itself, for example:

```
110 SYS 49057
```

The machine code must be in the memory if it is to be called up by SYS or the program may crash without a error message.

8.38 TRAP

Format: [linenumber TRAP linenumber2]

Arguments: linenumber2 is the linenumber of the first statement of the error treatment routine.

Abbreviation: tR

The TRAP statement uses BASIC to suspend the normal error treatment and activates the program to carry out its own treatment. Details on the increased possiblities for debugging are in Section 5.3.2. The statement in linenumber2 is executed if an error occurs. The statements for debugging should begin in linenumber2. The program for error treatment (debugging) can decide by using variables ER (error number) and EL (error line) what should be done for any error which may occur.

When an error occurs, ER contains the number of the error and EL the line number where it is to be found. Debugging of the ensuing errors is left to BASIC in the absence of the parameter linenumber2.

If TRAP is used in direct mode, error message ?ILLEGAL DIRECT appears.

Other statements used by error treatment are RESUME and DISPOSE. For example:

```
10     TRAP
20     REM PROGRAM START
30     ... BASIC statements ...
900    REM SUB PROGRAM FOR ERROR TREATMENT
910    REM ONLY FILE AND DEVICE ERROR
920    REM ARE TREATED, AS THE ER IS BETWEEN 1 AND 9
930    IF (ER < 1) OR (ER > 9) THEN GOTO 1010
940    PRINT "YOU HAVE DIFFICULTY WITH A FILE"
950    ...
1010   REM HERE OTHER ERRORS ARE TREATED
1020   ...
8000   END
```

TRAP without linenumber2 restores normal BASIC error processing (ie "resets" the trap).

8.39 WAIT

Format: (linenumber) WAIT address, mask1, mask2

Arguments: Address is the address of a memory location mask1 and mask2 are integer values.

Defaults: mask2 = 0

Abbreviation: wA

The WAIT statement continually checks the values in address until the condition described here is fulfilled. Then the next statement is executed. The WAIT statement is used to let the program pause whilst a certain value is being checked in the memory. WAIT statements are not used often; don't worry therefore if you don't understand everything immediately. Most programmers will never use this statement since it is normally used to survey the condition of an input channel.

Mask1 and mask2 are integer numbers and are compared with the memory byte at the point address. i.e. a mask can be used containing up to 8 ones or zeroes.

The WAIT statement functions as follows:

1. The values of address and mask2 are compared using the logical operation "Exclusive OR", if mask2 is given.

2. The result of the comparison is compared with mask1 using a logical "AND". If there is no mask2, the value of address is compared with mask1 using a logical "AND".

3. If the result of steps 1 and 2 is 0 (if all bits are "off") the WAIT statement is repeated.

4. If the result is not 0 (if one or more bits are "on") the next BASIC statement is activated.

The two masks are used as follows:

- mask1 filters out those bits which do not need to be checked. A bit which is 0 in mask1 will also produce a 0 in the result.
- mask2 switches bits round so that an "on" and an "off" condition can both be checked. A bit which needs to be checked for 0 must have a 1 at the corresponding point in mask2.

For example, if a program is to continue only if the far right hand bit at point 62255 is "off", then a 100 WAIT 62255,1,1 is used:

In this example, mask1 has the value 00000001 and mask2 00000001. The memory word at point 62255 has the value 145 (i.e. 10010001 in binary) and indicates the condition of an in/output channel. You must wait till the bit 0 (outer right) is "off". Then the following happens by using the WAIT statement:

1. The contents of 62255 is compared with an EOR to mask2:

```
62255          10010001
          EOR
mask2          00000001
Result1        10010000
```

2. The result is compared with mask1 by an AND:

```
Result1        10010000
          AND
mask1          00000001
Result2        00000000
```

3. The result is 0, so WAIT is executed again.

4. At some point the outer right hand bit in 62255 will be 0.

The WAIT statement reads the value in 62255. If the outer right hand bit is "off" the value of 62255 will be 10010000. This value is compared with mask2 in the first step of the WAIT statement. This means:

```
62255          10010000
          EOR
mask2          00000001
Result1        10010001
```

An AND comparison with mask1 is executed again:

```
Result1        10010001
          AND
mask1          00000001
Result2        00000001
```

Now the result is non-zero and the next statement after WAIT is executed.

Mask2 is not needed if it is only required to check that a bit is "on":

```
100 WAIT 62255, 1
```

No EOR is used during the execution of the statement. Mask2 is given 0, which does not alter a bit. The value of 62255 is compared to mask1 by AND. Assuming the value of 62255 is (00010000), the following happens:

```
62255          00010000
          AND
mask1          00000001
Result         00000000
```

The result is 0 which means that the outer right bit (checked because of mask1) is 0. WAIT is executed again and reads the value 62255 once more. If the value is now 145 (10010001), the following procedure takes place:

```
62255           10010001
          AND
mask1           00000001
Result          00000001
```

The result is non-zero, so that the statement following WAIT is executed.

It can be seen in the next example how to check if bit 4 is "off" or if bit 7 is "on". (Remember that mask2 is used to check if a bit is "off".)

```
100  WAIT 36548,144,16
```

The value 65 (01000001) is in memory location 36548. Bits 7 and 4 are both "off". Only bits 6 and 0 are "on". After carrying out step 1, bit 4 is switched by:-

```
36548           01000001
          EOR
mask2           00010000
Result1         01010001
```

Now the result is compared with mask1 by AND:

```
Result          01010001
          AND
mask1           10010000
Result2         00010000
```

The result is non-zero and the next statement is executed. Bit 4 was "off". Although bit 7 was not "on", WAIT established that bit 4 was "off" and continued the processing of the program.

Take care:- An endless loop can be produced with the WAIT statement.

WAIT cannot be interrupted using the STOP key!

CHAPTER 9

BASIC FUNCTIONS

ABS	POS
ASC	RIGHT$
ATN	RND
CHR$	SGN
COS	SIN
ERR$	SPC
EXP	SQR
FRE	ST
INSTR	STR$
INT	TAB
LEFT$	TAN
LEN	TI$
LOG	USR
MID$	VAL
PEEK	

The 700 series has a range of built-in functions incorporated in BASIC, and these can be used without further definition. The function parameter can be a number or a variable, (which can have a new value at each function call,) and is always enclosed in brackets.

Built-in functions can be used in both direct and program modes.

- Any variable name can be allocated to the function, for example:-

```
ARCTG = ATN((X*Y*Z)+(R/2))
NUM = VAL(S$)
```

- Functions of functions can be formed, as can expressions with more than one function, for example:-

```
RESULT = SQR(A*A+B*B) + COS(Y/4.777)
ANSW = LOG(ABS(INT(XX)))
```

- Functions can be used in direct mode, for example:-

```
?SQR(125.68)
?FRE(1)
```

The BASIC functions work with integer, real or text variables, depending on the function.

If a real number is given to a function which works with integers, the number is truncated. The following table contains value types into which the BASIC functions transfer the results.

The BASIC functions

	Result		Arguments		
Function	Numerical	Text	Numeric		String
ABS	X				X
ASC	X				X
ATN	X		X		
CHR$		X	X		
COS	X		X		
ERR$		X	X		
EXP	X		X		
FRE	X		X	or	X
INSTR	X		X	and	X
INT	X		X		
LEFT$		X	X	and	X
LEN	X				X
LOG	X		X		
MID$		X	X	and	X
PEEK	X		X		
POS	X		X		
RIGHT$		X	X	and	X
RND	X		X		
SGN	X		X		
SIN	X		X		
SPC		X	X		
SQR	X		X		
ST	X		X		
STR$		X	X		
TAB		X	X		
TAN	X		X		
TI$		X			X
USR	X		X		
VAL	X				X

9.1 ABS

Format: ABS (expression)

Arguments: expression is a numerical expression

Abbreviation: aB

The ABS function calculates the absolute value of a number. The absolute value is the positive value of expression. For example:-

```
PRINT ABS (7*(-35))                  ...Prints the value 245
10  PRINT ABS(1234)                  ...Prints the value 1234
A=20:B=-1: PRINT ABS(A*B)            ...Prints the value 20
D=-1:C=-9: PRINT ABS(C*D)            ...Prints the value 9
PRINT ABS(2*(-2.1))                  ...Prints the value 4.2
```

9.2 ASC

Format: ASC (expression)

Arguments: expression is a string expression

Abbreviation: aS

ASC returns the ASCII code of the first character in the expression. If expression is the null string, the error message ?ILLEGAL QUANTITY will appear. For example:-

```
10  X$ = "TEST"
20  PRINT ASC(X$)
RUN
84
READY
```

84 is the ASCII code for T. (See table in Appendix.)

9.3 ATN

Format: ATN (expression)

Argument: expression is a numerical expression

Abbreviation: aT

The ATN function calculates the arctangent of expression. The arctangent is given in radians. The range is from - /2 to + /2. As expression can be an integer or a real, the calculation is executed in floating point format. For example:-

```
10  INPUT X
20  PRINT ATN(X)
RUN
1.24904577
READY
```

9.4 CHR$

Format: CHR$ (expression)

Arguments: expression is an integer

The CHR$ function returns the character represented in the ASCII code be expression. (See Appendix on ASCII code.) Expression must be a number between 0 and 255.

The CHR$ function is the reverse function of the ASC function. For example:-

```
PRINT CHR$ (66)
B
READY
```

9.5 COS

Format:- COS (expression)

Argument: expression is a numerical expression

Abbreviation: none

The COS function calculates the cosine of expression. Expression is assumed to be in radians.

An integer or real number can be used for expression.

The calculation takes place in floating point format. For example:-

```
PRINT COS(5-1)
-.65364362
```

9.6 ERR$

Format: ERR$ (expression)

Argument: expression is a numerical expression

Abbreviation: eR

The ERR$ function returns the text of the standard error message whose number is expression. Expression must be a number between 0 and 42.

If ERR$ is used with a TRAP statement, standard error messages can be displayed. See Section 8.38 for TRAP.

Example:-

In this example it can be seen how the ERR$ function can be used together with the TRAP statement. The variable EL indicates the line number where the error occurred and ER is the error number whose more exact description is printed by way of the ERR$ function.

```
10    TRAP 1000
      ... BASIC statements ...
1000  REM THE ERRORS ARE ANALYSED.  IF THERE IS A SYNTAX
1010  REM ERROR, THE PROGRAM SHOULD BE STOPPED
1020  IF ER = 21 THEN PRINT EL, ERR$(ER): STOP
1030  REM IT IS NOT A SYNTAX ERROR.  THEN THE ERROR IS TESTED
1040  REM AND MESSAGE IS PRINTED
1050  IF ER = 9 THEN PRINT EL, ERR$(ER): RESUME 100
1060  IF ER = 30 THEN RESUME 150
      ... BASIC statements ...
1110  RESUME 975
1120  ... BASIC statements ...
```

9.7 EXP

Format: EXP (expression)

Argument: expression is a numerical expression

Abbreviation: eX

The EXP function calculates e (2.718281...) raised to the power expression. Expression must be in the range -88 to +88 approximately. If the EXP function causes an overflow, the error message ?OVERFLOW appears. The result of EXP() is always positive. For example:-

PRINT EXP(4)	Prints the value of the exponential of 4 to base e (about 54.6).

Note: EXP(0) is 1.

9.8 FRE

Format: FRE (expression)

Argument: expression is an integer or a string expression

Abbreviation: fR

The FRE function gives the number of free bytes which BASIC can use for program text, simple variables, arrays and strings in a memory bank.

The location which is available for these four areas (program, simple variables, arrays and strings) depends on the amount of memory the computer has available.

	128K	256K
BASIC program	1	1
Arrays	2	2
Simple variables	2	3
String variables	2	4

The value given by the FRE function depends on expression as follows:-

- If expression is a number, FRE gives the free bytes in the requested bank.
- If expression is a string expression, FRE gives the free memory available for string storage.

(FRE returns 0 if non-existent memory locations are called, or if the system bank is specified.) For example:-

```
10  N% = (FRE(2)-100)/5
20  DIM A(N%)
```

Here the memory available is determined with a FRE function before an array is defined.

9.9 INSTR

Format: INSTR (expression1, expression2, (expression3))

Arguments: expression1 and expression2 are string expressions
expression3 is a numerical expression

Default: expression3 = 1

Abbreviation: inS

The INSTR function locates a section of a string (i.e. it finds a substring). Expression2 is found in expression1. The search begins at the character specified by expression3 in string expression1. Expression3 must be between 1 and 255. If no number is given for expression 3, 1 is used. I.e. the whole of expression1 is searched.

- If expression2 is not found, INSTR has value 0
- If expression2 is found, INSTR gives the position of the first matching character. For example:-

```
10     A$ =  "MR MRS MISS MS"
20     ...read a name and check
60     IF INSTR(A$,B$) > 0 THEN GOSUB 1500: ELSE GOSUB 2000
       ... BASIC statements ...
1500   REM HERE THE CORRECT DATA SHOULD BE PROCESSED
       ... BASIC statements ...
2000   REM ERRORS IN NAME SHOULD BE PROCESSED HERE
       ... BASIC statements ...
```

9.10 INT

Format: INT (expression)

Argument: expression is a numerical expression

Abbreviation: None

The INT function calculated the largest INTEGER value which is smaller than or equal to the value in expression.

Examples:-

```
PRINT INT (1234.56)                    Prints the value 1234
PRINT INT (-1234.56)                   Prints the value -1235
```

9.11 LEFT$

Format: LEFT$ (expression1, expression2)

Arguments: expression1 is a string expression
expression2 is a numerical expression

Abbreviation: leF

Returns a substring from the left end of a string. Expression2 must be a number between 0 and 255.

If expression2 is larger than the length of expression1, the function returns the whole of expression1. (Use the LEN function to check.)

If expression2 is 0, the LEFT$ function returns a null string.

The LEFT$, MID$, and RIGHT$ functions can be used with the INSTR function for text processing. For example:-

```
10  A$ = "COMMODORE COMPUTER"
20  B$ = LEFT$(A$,9)
30  PRINT
RUN
COMMODORE
READY.
```

9.12 LEN

Format: LEN (expression)

Argument: expression is a string expression

Abbreviation: None

The LEN function returns the number of characters in expression (i.e. the lengths). The LEN function counts all characters in expression even those which are not printable or which are spaces. For example:-

```
10  X$ = "COMMODORE COMPUTER" + CHR$ (27): REM 27 IS NON PRINTING
20  PRINT LEN (X$)
RUN
19
READY
```

Note:

10 PRINTLEN("COMMODORE COMPUTER"~~)~~+CHR$(27)) would be equally acceptable.

9.13 LOG

Format: LOG (expression)

Argument: expression is a numerical expression

Abbreviation: None

The LOG function returns the natural logarithm (base e) of the expression. Expression must always be positive. For example:-

PRINT LOG (45/7) Prints the value 1.86075234

9.14 MID$

Format: MID$ (expression1, expression2, [expression3])

Arguments: expression1 is a string expression
expression2 and expression3 are integers

Default: expression3 is the number of all characters from character expression2 to end of string.

Abbreviation: mI

The MID$ function returns a substring containing expression3 characters from expression1 starting at the character at position expression2 onwards. Expression2 and expression3 must be between 0 and 255.

If there is no value given for expression3, or if there are fewer characters in expression1 than in expression3, then the function returns all characters from position expression2 to the end of the text.

If there is a number given for expression2 which is longer than expression1, then MID$ returns a null string. For example:-

```
PRINT "GOOD " MID$("MORNINGAFTERNOON",8,9)
```

Prints: GOOD AFTERNOON.

9.15 PEEK

Format: PEEK (address)

Argument: address is an integer

Abbreviation: pE

The PEEK function returns the decimal value of address. The value of address must be between 0 and 65535. The PEEK function returns a value between 0 and 255.

The PEEK function, together with the BANK statement, can reach any address in the memory. Details on BANK statement can be found in Section 8.1.

Example:-

```
10 PRINT PEEK (36879)
```

Prints the contents of the location 36879.

9.16 POS

Format: POS (dummy)

Argument: dummy is any number

Abbreviation: None

The POS function gives the point where the next character is to be printed. I.e. the position of the cursor. Any value can be given to dummy.

The cursor positions:-

- Far left is position 0
- Far right is position 79

Example:-

Here, a carriage return character is printed if the cursor is beyond location 20.

```
IF POS (X) > 20 THEN PRINT CHR$ (13) + A$: ELSE PRINT A$
```

9.17 RIGHT$

Format: RIGHT$ (expression1, expression2)

Arguments: expression1 is a string expression
expression2 is a numerical expression

Abbreviation: rI

The RIGHT$ function returns a substring of expression1 containing the number of characters specified by expression2. Expression2 must be a number between 0 and 255. If expression2 is longer than expression1, the RIGHT$ function will return the whole of expression1. If expression2 is equal to 0, RIGHT$ returns a null string. For example:-

```
10  A$ = "OFFICE MACHINE"
20  B$ = RIGHT$ (A$,11)
30  PRINT B$
RUN
ICE MACHINE
READY
```

9.18 RND

Format: RND (expression)

Argument: expression is a numerical expression

Abbreviation: rN

The RND function provides a random number between 0 and 1. The number does not actually occur randomly, but is calculated by the computer by an intricate algorithm (pseudo random number). To do this, there are two possibilities:-

- expression < 0
  The algorithm uses the expression number to calculate the random number ("seed").
- expression >= 0
  The algorithm uses the last previously formed random number to calculate the new random number ("series").

For example:-

Five random numbers are printed. (They are in the range 0 to 100).

```
10  FOR I = 1 TO 5
20  PRINT INT(RND(0)*100);
30  NEXT
RUN
24 30 31 51 5
READY
```

RUN again and 5 new random numbers are printed.

If you now add the program line:-

```
5  X = RND(-1)
```

Every program run will give the same sequence of "random" numbers, since line 5 now "seeds" the random number series.

9.19 SGN

Format: SGN (expression)

Argument: expression is a numerical expression

Abbreviation: sG

The SGN function returns the sign of expression. The values returned are as follows:- if X < 0 then SGN(X) = -1, if X = 0 then SGN(X) = 0, if X > 0 then SGN(X) = +1.
For example:-

```
ON SGN(X)+2 GOTO 100,200,300
```

This jumps to line 100 for X < 0, 200 for X = 0 or 300 for X > 0.

9.20 SIN

Format: SIN (expression)

Argument: expression is a numerical expression.

Abbreviation: sI

The SIN function calculates the sine of expression. Expression is assumed to be in Radians. An integer or a real number can be used for expression. The calculation takes place in the floating point format.

9.21 SPC

Format: SPC (expression)

Argument: expression is an integer expression.

Abbreviation: sP

The SPC function prints expression spaces. The value of expression must be between 0 and 255.

The SPC function can only be used as part of a PRINT statement. For example:-

```
PRINT"IN SECTION"SPC(20)POS(X)
IN SECTION                    30

READY.
```

9.22 SQR

Format: SQR (expression)

Argument: expression is a numerical expression

Abbreviation: sQ

The SQR function calculates the square root of expression. Expression must be larger than or equal to 0. For example:-

```
PRINT 10, SQR(10)
10        3.16227766

READY.
```

9.23 ST

Format: ST

The STATUS function returns the value of the reserved variable ST for the preceding input/ouput operation.

The value of the STATUS function depends on the operation and the device.

Function Values of STATUS function

STATUS bit Position	STATUS numerical Value	Meaning
0	1	Timeout output.
1	2	Timeout input.
6	64	End of file.
7	-128	Device not present.

For example:-

```
10  OPEN 2,8,2, "MASTER FILE,S"
20  GET#2, A$
30  IF STATUS AND 64 THEN 60
40  PRINT A$
50  GOTO20
60  PRINT A$:CLOSE 2
```

Here STATUS is used to check for the end of a file before closing it.

(Note: When using the RS232 interface, the ST has a different meaning.)

9.24 STR$

Format: STR$ (expression)

Argument: expression is a numerical expression

Abbreviation: stR

The STR$ function returns the ASCII text equivalent to expression. This is very useful if text is to be compiled from discrete characters or groups of characters, especially if the characters are numeric.

The VAL function (see 9.29) operates in the opposite way to STR$.

The length of the text returned depends on the value in expression. The length can be determined by using the LEN function. For example:-

```
PRINT "$" + STR$(2.77)              $ 2.77 is printed
PRINT STR$(15Ø)+".ØØ"                15Ø.ØØ is printed
```

9.25 TAB

Format: TAB (expression)

Argument: expression is an integer expression

Abbreviation: tA

The TAB function moves the cursor to the position indicated by expression. If the cursor is already beyond this point, TAB places the cursor in the next column.

Expression must be between Ø and 255, and columns are numbered starting at Ø at the left hand edge. For example:-

```
PRINT TAB(39) "123456"
```

Note: This command is always used as part of a PRINT command.

9.26 TAN

Format: TAN (expression)

Argument: expression is a numerical expression

Abbreviation: None

The TAN function calculates the tangent of expression. Expression is assumed to be Radians. Although expression can be an integer or real number, the calculation is always performed in floating point format. If the expression value causes an overflow, the error message ?OVERFLOW appears.

9.27 TI$

Format: TI$

The TI$ function returns the time from the internal clock. The string TI$ has 7 characters which are hours, minutes, seconds and tenths of seconds (HHMMSST). For example:-

```
10   TI$ = "0000000"
     ... BASIC statements ...
500  TA$ = TI$
510  T$ = LEFT$(TA$,2)+":"+MID$(TA$,3,2)+":"
520  PRINT T$ + MID$(TA$,5,2)+"."+RIGHT$(TA$,1)
```

Here the time is set to 0 and then, after the program run, the time elapsed is printed using TI$.

9.28 USR

Format: USR (expression)

Argument: expression is a numerical expression

Abbreviation: uS

The USR function calls up the assembler subprogram written by the user, the jump address of which is held in locations 3 and 4 of memory Bank 15. Expression is stored in the accumulator before the subprogram is called.

The function value is obtained from the accumulator location 71 Hex in Bank 15 as soon as the assembler subprogram has been executed and the BASIC program is running again. The address of the assembler subprogram must be poked into locations 3 and 4 in bank 15 before the USR function can be used. For example:-

```
10   REM REMEMBER THAT THE ADDRESS OF THE
20   REM ASSEMBLER SUBPROGRAM MUST BE ENTERED
30   REM BEFORE THE PROGRAM CAN BE CALLED UP
40   REM WITH A USR FUNCTION
50   BANK 15:POKE 3,0:POKE 4,4
100  B = 12.345
120  C = USR(B/2)
```

Here, a value is stored in the accumulator and then an assembler program is called up.

9.29 VAL

Format: VAL (expression)

Argument: expression is a string expression

The VAL function returns the numerical value of the string.

If the first character in expression is not +, -, $ or a number, then the VAL function will return the value 0.

The VAL function works in the opposite way to the STR$ function. For example:-

```
10  REM CHECK IF A STRING IS NUMERIC
15  REM IF NOT, THERE IS AN ERROR MESSAGE
20  IF VAL(A$) = 0 THEN 40
35  GOTO 500
40  PRINT "NO NUMERICAL VALUE.  THE VALUE IS";A$
```

Here, the VAL function is used to decide whether a string contains numbers or not before using it in an expression.

CHAPTER 10

THE MACHINE LANGUAGE MONITOR

For the user who needs to directly control memory or to work with machine language programs, the operating system has a monitor through which one can obtain important information on the internal state of the computer at any time.

In general this means the contents of registers and memory locations. All addresses and contents are displayed in a hexadecimal (hex) presentation. Hex numbers are identified here as normal by the preceding $ sign. For example:-

Hex		Decimal
$0A	=	10
$0F	=	15
$10	=	16
$FF	=	255

Thus, all register contents are two-digit, all addresses are four digit hex numbers.

In some commands, the memory segment's address must precede the address so that six-digit "long addresses" are the result. The monitor always uses the address in the current segment (bank) when using a four-digit address.

The monitor is activated by the command "bank 15:SYS60950"
:SYS 6

First of all, the register contents of the CPU and the actual interrupt pointer are displayed. The display might look like this:-

```
   PC  IRQ  SR AC XR YR SP
.; 0000 FBE9 00 00 00 00 F9
.
```

The meaning of this display is explained thus:-

PC:	Program counter, address of the next command to be carried out
IRQ:	Interrupt pointer ($300/301)
SR:	Status register
AC:	Accumulator
XR:	x-register
YR:	y-register
SP:	Stack pointer

The semicolon in the line with the register contents means that all values in this line may be altered; the changes are made when the RETURN key is pressed. The full stop at the beginning of the line indicates that the computer's monitor is running.

The following commands are valid for this mode:-

R

displays the register contents

M address [address]

displays the contents of the specified memory location (or all locations up to the second address).

The colon at the start of the line means that you can change the contents.

G [address]

Jumps to the main code program at the given address. If the address is missing, the microprocessor continues from the next command using the program counter.

L "NAME",device

Loads the program with the given name from the given device, (2-digit hex) into the preselected bank. No pointers are changed in the computer, unlike with the corresponding BASIC command.

S "NAME", device, longaddress, longaddress

Stores the memory contents between the given longaddresses as a file under the given name.

U device

Sets the default value for the disk device. (For use with @ and other commands.)

V segment

Selects the given bank for any following monitor commands. The selected segment (bank) can be determined at any time by m ØØØ1.

Z

Switches to any built-in co-processor (will "crash" the machine if there is no co-processor to accept control).

@ [command]

If this command is immediately followed by a RETURN, the computer displays the disk error message. The device address is normally 8 and the command channel 15. (See command U to change the device number.) If a command follows the @, then this command is transferred to the disk drive using the command channel. For example:-

@IØ

initialises drive Ø.

X

Warm starts BASIC once more. (I.e. exits the Machine Code Monitor and gives control back to BASIC.)

If a given command is not recognised an attempt is made to load a file of the same name from disk.
If this occurs successfully, the monitor jumps to the load start address.

** This process is only applicable to Bank 15. **

Note:- should the file not exist a kernal error message:-

I/O error#4 (file not found)

will be displayed, or if no disk drive is connected then:-

I/O error#5 (device not present)

will be displayed.

Further information about the Kernal is in the Kernal section of this manual.

APPENDIX A

The BASIC 4.0+ interpreter allows access to the memory of the computer. The size of the available memory depends on the computer model.

The following BASIC keywords are used with the memory:-

- BANK
- BLOAD
- BSAVE
- PEEK (a function)
- POKE

The BANK statement is the central element for accessing the multiple memory banks in the 700. The statement determines which bank will be dealt with by the BLOAD, BSAVE, POKE statements, and the PEEK function, normally dealt with by Bank 15.

When a BANK statement is used, all following BLOAD, BSAVE, POKE and PEEK operations refer to the newly defined bank.

The BLOAD statement is also used to load assembler subprograms from BASIC programs for special purposes.

Memory Organisation

The whole memory is divided into segments or banks. Each of these banks is an area of 64K bytes. A maximum of 16 of these banks can be resident. The banks are numbered from 0 to 15, ($00 to $0F).

Some banks have a fixed use which is partly dependent on the available memory.

In 128K models, it is distributed as follows:-

Bank 1: contains the BASIC text, i.e. the programs you use.
Bank 2: is used for variable storage.

In models with 256K capacity, Bank 1 is used in exactly the same way as for that in 128K versions, then ...

Arrays are stored in Bank 2.

Simple variables (non-indexed variables) are stored in Bank 3. (This bank also has space reserved for the disk operating system.)

Bank 4 contains the strings.

The application of Bank 15 is identical in all cases: The BASIC interpreter, the editor, the operating system, the input/output section and the system information (zeropage etc.) are to be found here.

The addresses from $2000 (8192) to $7fff (32767) are kept open; this is for any individual expansion. To this end, the address lines are available on the cartridge connector. If necessary, ROM modules, RAM memory or any input/output sections (all mixed) may be located here.

Memory Distribution in Segment (Bank) 15

Address (hexadecimal)

$FFFF
Kernal ROM (operating system)
$E000
Input/output section (see below)
$C000
BASIC ROM HI
$A000
BASIC ROM LO
$8000
Cartridge (Bank 3)
$6000
Cartridge (Bank 2)
$4000
Cartridge (Bank 1)
$2000
4K disk ROM
$1000
2K external buffer RAM
$0800
2K RAM — 0400 - 07FF → free RAM. / 0001 - 03FF → used for operating system
$0001 Indirect Register
$0000 Execute Register

I/O Section

$E000
TPI 6525 (keyboard)
$DF00
TPI 6525 (IEEE, User)
$DE00
ACIA 6551 (RS232)
$DD00
CIA 6526 (User, Inter-proc.)
$DC00
Free (co-proc.) Z-80, 8088
$DB00
SID 6581 (sound)
$DA00
Free (disk-input/output)
$D900
CRTC 6545 (screen)
$D800
Screen memory
$D000
Unused (reserved for character ROM in P128)
$C000

APPENDIX B

700 machines are equipped with an RS232 port as standard. The port is driven by an ACIA 6551 integrated circuit which is located between $DD00 and $DE00 in the system bank(15).

The MOS Technology Asynchronous Communication Interface Adapter 6551 allows for the following:-

- On-chip baud rate generating rates between 50 and 19,200 baud.
- Echo mode.
- False start bit detection.
- Bidirectional data.
- External non-standard clock input for baud rates up to 125,000 baud.
- Programmable word length.
- Programmable number of stop bits.
- Parity generation and detection. (Odd, even, none, mark and space are all useable.)
- Full or Half duplex.
- 5,6,7 or 8 bit transmission.

The Port it drives has the following pin connections:-

Pin	1	Shield		
	2	TxD	- Transmit data	output
	3	RxD	- Receive data	input
	4	RTS	- Ready to send	output
	5	CTS	- Clear to send	input
	6	DSR	- Data set ready	input
	7	Ground		
	8	DCD	- Data carry detect	input
	11	+5v.		
	18	-12v.		
	20	DTR	- Data terminal ready	output
	24	XMIT CLK	- Transmit clock	output/input

All other pins are not connected. See note 1 on Signals and note 8 on plug types.

Interrupts from the DCD and DSR lines are processed by the 6551 internal interrupt logic circuits. The 6551 can also generate an interrupt itself which is processed by the 6509 CPU in the 700. DTR and RTS lines are signalled by the 6551 command register logic.

The 6551 has five main registers:-

- TRANSMIT DATA register (TDR)
- STATUS register (SR)
- CONTROL register (CR)
- RECEIVE DATA register (RDR)
- COMMAND register (CMR)

Register Addresses

TDR/RDR	$DD00	*
SR	$DD01	
CR	$DD02	
CMR	$DD03	

* TDR if written to or RDR if read from.

TDR and RDR are used for temporary data storage. The SR is used to indicate the status of the various functions of the 6551 and may be interpreted as follows:-

Bit 0 :Parity error if set. Self clearing.
Bit 1 :Framing error if set. Self clearing.
Bit 2 :Overrun error if set. Self clearing.
Bit 3 :Receive data register full if set.
Bit 4 :Transmit data register empty if set.
Bit 5 :DCD line in high logic state if set.
Bit 6 :DSR in high logic state if set.
Bit 7 :(IRQ) interrupt if set.

It can be seen from the above that a status register containing 0 indicates that all is well. See also note 5.

CR and CMR are set from the BASIC statement OPEN.

The OPEN statement has the following format:-

OPEN channelnumber,2,secondaryaddress,bytestring

- The channelnumber may be any number between 0 and 255. If a channelnumber greater than 127 is chosen then CR and LF are sent with each PRINT#, otherwise CR alone is sent (see note 3).
- 2 is the primary address of the RS232 port.
- The secondaryaddress of the RS232 port may be one of the following six numbers according to your requirements:-

1 for Transmit characters only.
2 for Receive characters only.
3 for Transmit and Receive characters.
129 for Transmit and convert characters.
130 for Receive and convert characters.
131 for Transmit, Receive and convert characters.

(Conversion is from CBM to ASCII and vice versa.)

- The bytestring contains four bytes/characters and is composed as follows:-

The First byte is the Control Register byte. The Second byte is the Command Register byte. The Third and Fourth bytes are not used in a 700, but dummy characters must be sent to the 6551 or errors will occur. For example:- send "++".

The CR byte controls the speed of transmission, the number of stop bits and the word length:-

The bits 0 and 3 are used as follows:-

Bit 3 2 1 0	Baud Rate	Decimal value
0 0 0 0	External rate x 16	0 *
0 0 0 1	50	1
0 0 1 0	75	2
0 0 1 1	109.92	3
0 1 0 0	134.58	4
0 1 0 1	150	5
0 1 1 0	300	6
0 1 1 1	600	7
1 0 0 0	1200	8
1 0 0 1	1800	9
1 0 1 0	2400	10
1 0 1 1	3600	11
1 1 0 0	4800	12
1 1 0 1	7200	13
1 1 1 0	9600	14
1 1 1 1	19200	15

* Receive only.

Bit 4 should be 1 unless the external clock is being used. (Decimal value 16.)

Bits 5 and 6 are used as follows:-

Bit 6 5	Word length	Decimal value
0 0	8	0
0 1	7	32
1 0	6	64
1 1	5	96

Bit 7 controls the number of stop bits and should be 0 for 1 stop bit, and 1 for all other purposes:-

- 2 stop bits
- 1 stop bit for 8 bit transmission (i.e. 8 bits and parity)
- 1.5 stop bits for 5 bit words without priority.

The CMR byte controls the handshake, duplex and parity options. (See note 4):-

Bit 0 controls the handshake line (DTR). If this bit is set (i.e. 1) then DTR is low logic and all interrupts are enabled along with the receiver. If not set then the receiver and all interrupts are disabled and DTR is high logic. All this implies "X-line" if this bit is on and "3-line" if it is off.

Bits 1,2 and 3 should be set to 0 for all purposes. (See note 4 for their purpose in the 6551).

Bit 4 sets "normal receiver"/Full duplex mode for the receiver when it is off (0). When on (decimal value 16), it sets "echo"/half duplex mode for the receiver.

Bits 5,6 and 7 control parity:-

Bit 7 6 5	Value	Parity mode	Comment
0 0 0	0	disabled	No bit generated/received
0 0 1	32	odd	Transmitter and Receiver
0 1 0	64	disabled	-
0 1 1	96	even	Transmitter and Receiver
1 0 0	128	disabled	-
1 0 1	160	mark	Mark parity bit transmitted
1 1 0	192	disabled	-
1 1 1	224	space	Space parity bit transmitted

Mark and Space modes disable the parity check.

Note 1) Interface signals:-

1a) The TxD output line is used to transfer serial data to the RS232 peripheral. The LSB (least significant bit) of the TDR (transmit data register) is the first data bit transmitted at the selected baud rate.

1b) The RxD input line is used to transfer serial data into the ACIA from the RS232 peripheral, LSB first. Baud rate is as selected or according to an externally generated receiver clock - see CR.

1c) The RxC (receive clock) line is used to indicate the Baud rate (x16), or clock rates, being used by the ACIA to clock the input data. When the interanl Baud rate generator is used this line supplies the clock being used (Baud rate x 16). When an external clock is being used, Baud rate option = 0000, this line is used to input the external clock (Baud rate x 16).

1d) The RTS output line is used to conrol the RS232 peripheral. The logic state of this line is determined by CMR.

1e) The CTS input line is used to control the transmitter. The transmitter is enabled if CTS is low logic, or if the CTS line is high, the transmitter is automatically disabled.

1f) The DTR output line is used to indicate the status of the ACIA to the RS232 peripheral. A high logic state means that the ACIA is disabled. A low logic state means that the ACIA is enabled. The 700 CPU (6509) controls this line through the CMR.

1g) The DSR input line is used to indicate the status of the RS232 peripheral to the ACIA, low logic means "ready" and high logic means "not ready", but the DSR must be connected. Even if the DSR is unused it must be driven high or low, (but not switched). If interrupts are enabled (see CMR bit 0) and a change in the logic state of DSR occurs, an interrupt will be signalled to the 6509 and bit 6 of SR (status register) will reflect the logic level or DSR. The state of DSR does not affect the transmitter or receiver operation directly, only signals from the 6509 (sent as a result of the interrupt generated by the ACIA) affect the operation.

1h) The DCD input line is used to signal the presence (or absence) of a carrier signal at the RS232 peripheral (normally used with modems). High logic means that a carrier signal is present and low logic means that it is not. Like DSR this line must be driven (see DSR). Similarly, if interrupts are enabled, IRQ is sent to the 6509 and bit 5 of SR reflects the logic level of DCD. DCD must be ow for the receiver to operate. Transmitter is only indirectly affected, if at all.

1i) DTR and CTS are not used (i.e. ignored) in "3-line" mode.

Note 2) Reset of the ACIA - see also note 5.

2a) Hardware reset (power on for example) sets all bits in CR and CMD to zero, sets bits 0,1,2,3 and 7 of SR to zero, and sets bit 4 of SR (TDR empty) to 1.

2b) Software reset (CLOSE command for example) sets bits 0,1,2,3, and 4 of CMR to zero, and sets bit 2 of SR (no overrun error) to zero.

All other bits of CR, CMR and SR are unaffected, except by direct intervention from the 6509.

Note 3) Channelnumber parameter in OPEN

If bit 7 of the channel number (logical file number) is low (i.e. channelnumber is less than 128) then PRINT# statements only send a CR (carriage return) character (chr$(13)). If bit 7 is high then CRLF (carriage return line feed) characters (chr$(13) + chr$(10)) are sent.

Note 4) CMR byte bits 1,2 and 3

These bits control Receiver interrupts and transmitter control interrupts. The 700 BASIC OPEN statement should not pass these bits and therefore they should be set to 0. However, their meaning in the ACIA is as follows:-

Bit 1 disables receiver interrupts if set (2), or enables receiver interrupts from bit 3 of SR (RDR full) if not set (0).

Bits 2 and 3 (transmitter controls):-

Bit 3 2	Value	Transmitter IRQ	RTS logic	Transmitter
0 0	0	Disabled	High	Off
0 1	4	Enabled	Low	On
1 0	8	Disabled	Low	On
1 1	12	Disabled	Low	Transmit BRK

Note 5) SR

Self clearing bits are cleared when error free data is next received. Bits 5 and 6 reflect the logic state of DCD and DSR and are not resettable.

Note 6) RS232 buffer.

The BASIC OPEN statement allocates a 256 byte buffer for the RS232. The statement does not perform a CLR however. (Unlike on the 64, for example.)

The BASIC CLOSE statement de-allocates the buffer. The buffer will be de-allocated regardless of its content, so you should read/send all the characters before CLOSEing the RS232 file.

It is often advisable to OPEN an RS232 file at the beginning of a program and leave it open until the program ends or has no further use for the RS232 peripheral.

Note 7) Technical.

7a) If you use an RS232 Modem, the 700 is normally configured to act as a "data terminal".

7b) The RS232 interface operates in an asynchronous manner. This means that the TxD line is kept high until characters are to be transmitted. (As opposed to synchronous operation where a fill character is passed when no characters are being transmitted.)

7c) The RS232 interface operates serially. This means that bits are sent on one data line one after another. (As opposed to parallel operation where eight bits are passed simultaneously on eight separate data lines.)

When a byte is to be sent serially the following occurs on the data line-

1) A start bit is sent (low logic) - The receiver uses this bit to synchronise itself with the transmitter.
2) The bits of information (LSB first) are sent.
3) The parity bit, if required is sent.
4) One or two stop bits are sent. (High logic.)
5) The line remains high logic and passive until the next byte is to be sent. The receiver waits.

Note 8) Plugs for peripheral connection

Cannon	CCITT V24	EIA	DIN 66	ID
1	1	AA	101	GND/E
2	2	BA	103	TxD
3	3	BB	104	RxD
4	4	CA	105	RTS
5	5	CB	106	CTS/RFS
6	6	CC	107	DSR
7	7	AB	102	SIG.GND
8	8	CF	109	DCD
20	20	CD	108/2	DTR
24	24	-	-	RxC

Example of an RS232 OPEN command.

```
OPEN 1,2,3,CHR$(6+16+96+128)+CHR$(1+16)+"++"
```

- channelnumber is 1, so PRINT# will use this channel.
- primary address is 2, the RS232 port.
- secondaryaddress is 3, enabling transmit/receive without conversion.
- CHR$(246) is the CR byte composed thus:-

```
  6  for 300 baud
 16  for Internal clock
 96  for 5 bit word
128  for 1.5 stop bits
```

- CHR$(17) is the CMR byte composed thus:-

```
 1  for X-line handshake
16  for full duplex
(No parity for 5 bit, 1.5 stop bit)
```

Another example.

```
OPEN 6,2,129,CHR$(24)+CHR$(112)+"++"
```

- channelnumber is 6, hence PRINT#6.
- secondaryaddress 129 converts and transmits.
- CR Byte enables 1200 baud, 8 bit word + 1 stop bit.
- CMR Byte allows for 3 line, half duplex, even parity.

Summary of the CR and CMR bytes.

CR byte = CHR$(A+B+C+D) where:-

A is a number between 0 and 15 for baud rate.
B is normally 16, but may be 0 for an external clock.
C is 0, 32,64 or 96 for word length.
D is 0 or 128 for stop bits.

CMR byte = CHR$(E+F+G+H) where:-

E is 0 or 1 for handshake.
F is 0 almost always. (See note 4 above.)
G is 0 or 16 for duplex.
H is 0, 32, 64, 96, 128, 160, 192 or 224 for parity.

Last words on RS232

Read the User Guide or Manual that comes with the RS232 peripheral you intend to connect to the 700. It is important that you fully understand the way the RS232 is configured for your peripheral.

This section on the RS232 and the ACIA requires careful reading to ensure good results.

A program example is as follows:-

```
 10  trap80:print"<CLR>RS232 input appears in normal video.<DOWN>"
 20  print"Keyboard output appears in <RVS>reverse<OFF>
     video.<DOWN>"
 30  open1,2,3,chr$(246)+chr$(17)+"++"
 40  get#1,a$:ifa$=""then60:elseifx=1thenprint:x=0
 50  printa$;:goto40
 60  getb$:ifb$=""then40:elseifx=0thenprint:x=1
 70  print"<RVS>"b$"<OFF>";:print#1,b$:goto40
 80  ifel=30thenprint"<DOWN>ERROR in Open statement on
     line 30:-<DOWN>":list30
 90  ifel=0orer=14thenclose1:print:print"Stopped.":end
100  print"<DOWN>"err$(er)" in line"el:".. ST="st:end
```

KEY:-

<RVS> means reverse video on.
<OFF> means reverse video off.
<DOWN> means cursor down.
<CLR> means clear screen.

APPENDIX C

This section gives the key numbers which you use in your sound program, based on the three voices.

To set sound control with BASIC, you need commands of the form:-

```
POKE (register),(content)
```

You must add all the required values in the split registers, for example:-

For average rise, average decay in Voice 2:-

```
BANK 15
POKE 55808 + 12,5*16+7  (or POKE 55820,87)
```

base address + register, attack + decay

Take care that you set the volume before producing a tone. POKE 55832 followed by a number between 0 and 15 sets the volume for all three voices.

Control Register for Tone Production

The Base address of SID in Bank 15 is 55808 (Dec) = DA00 (Hex)

Register: Voice 1	Voice 2	Voice 3	Content	
0	7	14	Frequency, lo-byte (0...255)	
1	8	15	Frequency, hi-byte (0...255)	
2	9	16	Pulse ratio, lo-byte (0...255) Only for square	
3	10	17	Pulse ratio, hi-byte (0...15) Only for square	
4	11	18	Wave form: Noise 129, Square 65, Sawtooth 33, Triangle 17	
5	12	19	Attack 0*16(hard)..15*16(soft)	Decay 0(quick)..15(slow)
6	13	20	Sustain 0*16(silent)..15*16(loud)	Release 0(quick)..15(slow)
24	24	24	Volume 0(silent)..15(full volume)	

For example:-

Continuous tone (Note C5) on Voice 2, (triangle waveform)

```
SI=55808
BANK 15
POKE SI+24,15:POKE SI+7,37:POKE SI+8,17:POKE SI+13,240
(Volume):(Frequency, Lo):(Frequency, Hi):(Sustain level, 15*16)
Switch tone on: POKE SI+11,17
Switch off:     POKE SI+11,0
```

Other SID Registers

Register	Content						
21	Filter frequency, Lo-byte (0...7)						
22	Filter frequency, Hi-byte (0...255)						
23	Resonance	& Filter source					
	0(none)...15*16(strong)	External	Voice3	Voice2	Voice1	None	
		8	4	2	1	0	
24	Filter mode	& Volume					
	(See note)	High pass	Band pass	Low pass			
	128	64	32	16	0(silent)...15(loud)		

Note: This isolates voice 3 so that it may be used to generate effects without being output itself.

The SID also has two further registers:-

Register	Content
27	Oscillator 3
28	Envelope 3

The momentary value of the oscillator and the envelope generator of voice 3 can be read in registers 27 and 28.

These are used for example, to produce random generators or to influence the other voices with these values, in order to achieve special sound effects.

Using these settings, you can imitate various musical instruments

Instrument	Waveform		Attack	Sustain
Piano	Pulse	65	9	0
Flute	Triangle	17	96	0
Cymbals	Sweep	33	9	0
Xylophone	Triangle	17	9	0
Organ	Triangle	17	0	240
Accordeon	Triangle	17	102	0
Trumpet	Sweep	33	96	0

Note: The settings for the envelope should always be POKEd before the waveforms are POKEd.

APPENDIX D

Below you will find a complete list of the notes, frequencies, frequency parameters, and the values which must be POKEd into the sound chip registers FREQ HI and FREQ LO in order to produce the required tone.

You are not bound by the values in this table! If you are using several voices, you can even consciously "mistune" the second and third voices, i.e. slightly(!) change the Lo-Byte in the table. This will result in a fuller sound.

No	Note-octave	Frequency(Hz)	Parameter	Hi-byte	Lo-byte
0	C-0	16.4	137	0	137
1	C#-0	17.3	145	0	145
2	D-0	18.4	154	0	154
3	D#-0	19.4	163	0	163
4	E-0	20.6	173	0	173
5	F-0	21.8	183	0	183
6	F#-0	23.1	194	0	194
7	G-0	24.5	205	0	205
8	G#-0	26.0	218	0	218
9	A-0	27.5	231	0	231
10	A#-0	29.1	244	0	244
11	Bb-0	30.9	259	1	3
12	C-1	32.7	274	1	18
13	C#-1	34.6	291	1	35
14	D-1	36.7	308	1	52
15	D#-1	38.9	326	1	70
16	E-1	41.2	346	1	90
17	F-1	43.7	366	1	110
18	F#-1	46.2	388	1	132
19	G-1	49.0	411	1	155
20	G#-1	51.9	435	1	179
21	A-1	55.0	461	1	205
22	A#-1	58.3	489	1	233
23	Bb-1	61.7	518	2	6
24	C-2	65.4	549	2	37
25	C#-2	69.3	581	2	69
26	D-2	73.4	616	2	104
27	D#-2	77.8	652	2	140
28	E-2	82.4	691	2	179
29	F-2	87.3	732	2	220
30	F#-2	92.5	776	3	8
31	G-2	98.0	822	3	54
32	G#-2	103.8	871	3	103
33	A-2	110.0	923	3	155
34	A#-2	116.5	977	3	209
35	Bb-2	123.5	1036	4	12
36	C-3	130.8	1097	4	73
37	C#-3	138.6	1162	4	138
38	D-3	146.8	1231	4	207
39	D#-3	155.6	1305	5	25
40	E-3	164.8	1382	5	102
41	F-3	174.6	1464	5	184
42	F#-3	185.0	1552	6	16
43	G-3	196.0	1644	6	108
44	G#-3	207.7	1742	6	206
45	A-3	220.0	1845	7	53
46	A#-3	233.1	1955	7	163
47	Bb-3	246.9	2071	8	23
48	C-4	261.6	2194	8	146
49	C#-4	277.2	2325	9	21
50	D-4	293.7	2463	9	159
51	D#-4	311.1	2609	10	49
52	E-4	329.6	2765	10	205
53	F-4	349.2	2929	11	113
54	F#-4	370.0	3103	12	31

55	G-4	392.0	3288	12	216
56	G#-4	415.3	3483	13	155
57	A-4	440.0	3690	14	106
58	A#-4	466.2	3910	15	70
59	Bb-4	493.9	4142	16	46
60	C-5	523.3	4389	17	37
61	C#-5	554.4	4649	18	41
62	D-5	587.3	4926	19	62
63	D#-5	622.3	5219	20	99
64	E-5	659.3	5529	21	153
65	F-5	698.5	5858	22	226
66	F#-5	740.0	6206	24	62
67	G-5	784.0	6575	25	175
68	G#-5	830.6	6966	27	54
69	A-5	880.0	7381	28	213
70	A#-5	932.3	7819	30	139
71	Bb-5	987.8	8284	32	92
72	C-6	1046.5	8777	34	73
73	C#-6	1108.7	9299	36	83
74	D-6	1174.7	9852	38	124
75	D#-6	1244.5	10438	40	198
76	E-6	1318.5	11058	43	50
77	F-6	1396.9	11716	45	196
78	F#-6	1480.0	12413	48	125
79	G-6	1568.0	13151	51	95
80	G#-6	1661.2	13933	54	109
81	A-6	1760.0	14761	57	169
82	A#-6	1864.7	15639	61	23
83	Bb-6	1975.5	16569	64	185
84	C-7	2093.0	17554	68	146
85	C#-7	2217.5	18598	72	166
86	D-7	2349.3	19704	76	248
87	D#-7	2489.0	20876	81	140
88	E-7	2637.0	22117	86	101
89	F-7	2793.8	23432	91	136
90	F#-7	2960.0	24825	96	249
91	G-7	3136.0	26301	102	189
92	G#-7	3322.4	27865	108	217
93	A-7	3520.0	29522	115	82
94	A#-7	3729.3	31278	122	46
95	Bb-7	3951.1	33138	129	114
96	C-8	4186.0	35108	137	36
97	C#-8	4434.9	37196	145	76
98	D-8	4698.6	39408	153	240
99	D#-8	4978.0	41751	163	23
100	E-8	5274.0	44234	172	202
101	F-8	5587.7	46864	183	16
102	F#-8	5919.9	49651	193	243
103	G-8	6271.9	52603	205	123
104	G#-8	6644.9	55731	217	179
105	A-8	7040.0	59045	230	165
106	A#-8	7458.6	62556	244	92

($90-95)

APPENDIX E

```
0000             * =$0000
0000     ;----------------------------------------------------------
0000     ; 6509  used to extend memory on bc2 & p2 systems
0000     ;   location - used to direct
0000     ;   $0000 - execution register (4 bits)
0000     ;   $0001 - indirect register (4 bits)
0000     ;
0000     ;   these registers provide 4 extra high-order address
0000     ;   control lines.  On 6509 reset all lines are high.
0000     ;
0000     ; current memory map:
0000     ;   segment 15- $ffff-$e000  rom (kernal)
0000     ;               $dfff-$df00  i/o 6525 tpi2
0000     ;               $deff-$de00  i/o 6525 tpi1
0000     ;               $ddff-$dd00  i/o 6551 acia
0000     ;               $dcff-$dc00  i/o 6526 cia
0000     ;               $dbff-$db00  i/o unused (z80,8088,68008)
0000     ;               $daff-$da00  i/o 6581 sid
0000     ;               $d9ff-$d900  i/o unused (disks)
0000     ;               $d8ff-$d800  i/o 6566 vic/ 6845 80-col
0000     ;               $d7ff-$d400  colour nybles/80-col screen
0000     ;               $d3ff-$d000  video matrix/80-col screen
0000     ;               $cfff-$c000  character dot rom (p2 only)
0000     ;               $bfff-$8000  roms external (language)
0000     ;               $7fff-$4000  roms external (extensions)
0000     ;               $3fff-$2000  rom external
0000     ;               $1fff-$1000  rom internal
0000     ;               $0fff-$0800  unused
0000     ;               $07ff-$0002  ram (kernal/basic system)
0000     ; segment 14 -  segment 8 open (future expansion)
0000     ; segment  7 -  $ffff-$0002  ram expansion (external)
0000     ; segment  6 -  $ffff-$0002  ram expansion (external)
0000     ; segment  5 -  $ffff-$0002  ram expansion (external)
0000     ; segment  4 -  $ffff-$0002  ram b2 expansion (p2 external)
0000     ; segment  3 -  $ffff-$0002  ram expansion
0000     ; segment  2 -  $ffff-$0002  ram b2 standard (p2 optional)
0000     ; segment  1 -  $ffff-$0002  ram b2 p2 standard
0000     ; segment  0 -  $ffff-$0002  ram p2 standard (b2 optional)
0000     ;
0000     ; the6509 registers appear in locations $0000 and
0000     ; $0001 in all segments of memory.
0000     ;
0000     ;----------------------------------------------------------
0000     e6509    *=*+1         ;6509 execution register
0001     i6509    *=*+1         ;6509 indirection register
0002     irom     =$f           ;indirect=rom or execution=rom
0002              * =$90
0090     ;kernal page zero variables
0090     ;
0090     ;kernal indirect address variables
0090     ;
0090     fnadr    *=*+3         ;address of file name string
0093     sal      *=*+1         ;current load/store address
0094     sah      *=*+1
0095     sas      *=*+1
```

(ie. select BANK 15)

($96 - be)

```
0096    eal      *=*+1        ;end of load/save
0097    eah      *=*+1
0098    eas      *=*+1
0099    stal     *=*+1        ;start of load/save
009a    stah     *=*+1
009b    stas     *=*+1
009c    ;
009c    ;frequently used kernal variables
009c    ;
009c    status   *=*+1        ;i/o operation status
009d    fnlen    *=*+1        ;file name length
009e    la       *=*+1        ;current logical index
009f    fa       *=*+1        ;current first address
00a0    sa       *=*+1        ;current second address
00a1    dfltn    *=*+1        ;default input device
00a2    dflto    *=*+1        ;default output device
00a3    ;
00a3    ;tape buffer pointer
00a3    ;
00a3    ;tapel   *=*+3        ;address of tape buffer
00a6    ;
00a6    ;rs232 buffer pointers
00a6    ;
00a6    ribuf    *=*+3        ;input buffer
00a9    ;
00a9    ;variables for kernal speed
00a9    ;
00a9    stkey    *=*+1        ;stop key flag
00aa    ctemp                 ;used to reduce cassette read times
00aa    c3po     *=*+1        ;ieee buffer flag
00ab    snswl                 ;used to reduce cassette read times
00ab    bsour    *=*+1        ;ieee character buffer
00ac    ;
00ac    ;   cassette temps - overlays ipc buffer
00ac    ;
00ac    ipoint                ;next 2 bytes used for transx code
00ac    syno     *=*+1
00ad    dpsw     *=*+1
00ae    ;   next 18 bytes also used for monitor
00ae    ptrl     *=*+1        ;index to pass1 errors
00af    ptr2     *=*+1        ;index to pass2 errors
00b0    pcntr    *=*+1
00b1    firt     *=*+1
00b2    cntdn    *=*+1
00b3    shcnl    *=*+1
00b4    rer      *=*+1
00b5    rez      *=*+1
00b6    rdflg    *=*+1
00b7    flagtl                ;temp during bit read time
00b7    shcnh    *=*+
00b8    cmp0     *=*+1
00b9    diff     *=*+1
00ba    prp      *=*+1
00bb    ochar    *=*+1
00bc    prty     *=*+1
00bd    fsblk    *=*+1
00be    mych     *=*+1
```

($bf – 300)

```
00bf    cdata    *=*+1        ;how to turn cassette timers on
00c0    ;screen editor page zero variables
00c0    ;
00c0    ;editor indirect address variables
00c0    ;
00c0             *=$c0        ;leave some space
00c0    pkybuf   *=*+2        ;start adr of pgm key
00c2    keypnt   *=*+2        ;current pgm key buf
00c4    sedsal   *=*+2        ;scroll ptr
00c6    sedeal   *=*+2        ;scroll ptr
00c8    pnt      *=*+2        ;current character pointer
00ca    ;
00ca    ;editor variables for speed and size
00ca    ;
00ca    tblx     *=*+1        ;cursor line
00cb    pntr     *=*+1        ;cursor column
00cc    grmode   *=*+1        ;graphic/text mode flag
00cd    lstx     *=*+1        ;last character index
00ce    lstp     *=*+1        ;screen edit start position
00cf    lsxp     *=*+1
00do    crsw     *=*+1        ;
00d1    ndx      *=*+1        ;index to keyd queue
00d2    qtsw     *=*+1        ;quote mode flag
00d3    insrt    *=*+1        ;insert mode flag
00d4    config   *=*+1        ;cursor type / char before blink (petii)
00d5    indx     *=*+1        ;last byte position on line (##234-02
00d6    kyndx    *=*+1        ;count of program key string    ##244-02)
00d7    rptcnt   *=*+1        ;delay tween chars
00d8    delay    *=*+1        ;delay to next repeat
00d9    ;
00d9    sedt1    *=*+1        ;frequently used temp variables
00da    sedt2    *=*+1
00db    ;
00db    ;frequently used editor variables
00db    ;
00db    data     *=*+1        ;current print data
00dc    sctop    *=*+1        ;top screen 0-25
00dd    scbot    *=*+1        ;bottom 0-25
00de    sclf     *=*+1        ;left margin
00df    scrt     *=*+1        ;right margin
00e0    modkey   *=*+1        ;keyscanner shift/control flags
                                                    ($ff-nokey)
00e1    norkey   *=*+1        ;keyscanner normal key number ($ff-nokey)
00e2    ;
00e2    ; see screen editor listings for usage in this area
00e2    ;
00e2             *=$f0        ;free zero page space, 16 bytes
00f0             *=$100       ;system rack area
0100    bad      *=*+1        ;cassette bad address table
0101             *=$1ff
01ff    stackp   *=*+1        ;system stack pointer tranx code
0200             *=$200
0200    buf      *=*+256      ;basic's rom page work area
0300    ;
0300    ;system ram vectors
0300    ;
0300    cinv     *=*+2        ;irq vector
```

($302 - 036c)

```
0302    cbinv    *=*+2        ;brk vector
0304    nminv    *=*+2        ;nmi vector
0306    iopen    *=*+2        ;open file vector
0308    iclose   *=*+2        ;close file vector
030a    ichkin   *=*+2        ;open chn in vector
030c    ickout   *=*+2        ;open chn out vector
030e    iclrch   *=*+2        ;close channel  vector
0310    ibasin   *=*+2        ;input from chn vector
0312    ibsout   *=*+2        ;output to chn vector
0314    istop    *=*+2        ;check stop key vector
0316    igetin   *=*+2        ;get from queue vector
0318    iclall   *=*+2        ;close all files vector
031a    iload    *=*+2        ;load from file vector
031c    isave    *=*+2        ;save to file vector
031e    usrcmd   *=*+2        ;monitor extension vector
0320    escvec   *=*+2        ;user esc key vector
0322    ctlvec   *=*+2        ;unused control key vector
0324    isecnd   *=*+2        ;ieee listen secondary address
0326    itksa    *=*+2        ;ieee talk secondary address
0328    iacptr   *=*+2        ;ieee character in routine
032a    iciout   *=*+2        ;ieee character out routine
032c    iuntlk   *=*+2        ;ieee bus untalk
032e    iunlsn   *=*+2        ;ieee bus unlisten
0330    ilistn   *=*+2        ;ieee listen device primary address
0332    italk    *=*+2        ;ieee talk device primary address
0334    ;
0334    ;kernal absolute variables
0334    ;
0334    lat      *=*+10       ;logical file numbers
033e    fat      *=*+10       ;device numbers
0348    sat      *=*+10       ;secondary addresses
0352    ;
0352    ;
0352    lowadr   *=*+3        ;start of system memory
0355    hiadr    *=*+3        ;top of system memory
0358    memstr   *=*+3        ;start of user memory
035b    memsiz   *=*+3        ;top of user memory
035e    timout   *=*+1        ;ieee timeout enable
035f    verck    *=*+1        ;load/verify flag
0360    ldtnd    *=*+1        ;device table index
0361    msgflg   *=*+1        ;message flag
0362    bufpt    *=*+1        ;cassette buffer index
0363    ;
0363    ;kernal temporary (local) variable
0363    ;
0363    t1       *=*+1
0364    t2       *=*+1
0365    xsav     *=*+1
0366    savx     *=*+1
0367    svxt     *=*+1
0368    temp     *=*+1
0369    alarm    *=*+1        ;irq variable holds 6526 irq's
036a    ;
036a    ;kernal cassette variables
036a    ;
036a    itape    *=*+2        ;indirect for cassette code
036c    cassvo   *=*+1        ;cassette read variable
```

(#036d - 0990

```
036d    aservo   *=*+1          ;flagt1***indicates t1 timeout cassette
                                 read
036e    caston   *=*+1          ;how to turn on timers
036f    relsal   *=*+1          ;moveable start load addr
0370    relsah   *=*+1
0371    relsas   *=*+1
0372    oldinv   *=*+3          ;restore user irq and i6509 after
                                 cassettes
0375    casl     *=*+1          ;cassette switch flag
0376    ;
0376    ;re232 information storage
0376    ;
0376    m51ctr   *=*+1          ;6551 control image
0377    m51cdr   *=*+1          ;6551 command image
0378             *=*+2
037a    rsstat   *=*+1          ;perm. rs232 status
037b    dcdsr    *=*+1          ;last dcd/dsr value
037c    ridbs    *=*+1          ;input start index
037d    ridbe    *=*+1          ;input end index
037e    ;
037e    ;screen editor absolute
037e    ;
037e             *=$380         ;block some area for editor
0380    pkyend   *=*+2          ;program key buffer end address
0382    pagsav   *=*+1          ;temp ram page
0383    ;
0383    ; see screen editor listings for other variables
0383    ;
0383             *=$3c0         ;free absolute space start
03c0    ;
03c0    ; system warm start variables and vectors
03c0    ;
03c0             *=$3f8
03f8    evect    *=*+5
03fd    warm     =$a5           ;warm start flag
03fd    winit    =$5a           ;initialization complete flag
03fd             *=$400
0400    ramloc
0400    ;
0400    ; kernal inter-process communication variables
0400             *=$0800
0800    ipbsiz = 16            ;ipc buffer size
0800    ;
0800    ; ipc buffer offsets
0800    ;
0800    ipccmd = 0              ;ipc command
0800    ipcjmp = 1              ;ipc jump address
0800    ipcin  = 3              ;ipc #input bytes
0800    ipcout = 4              ;ipc #output bytes
0800    ipcdat = 5              ;ipc data buffer (8 bytes max)
0800    ;
0800    ipb      *=*+ipbsiz     ;ipc buffer
0810    ipjtab   *=*+256        ;ipc jump table
0910    ipptab   *=*+128        ;ipc param spec table
0990    ;
0990             .end
0990             .lib scrn-declare
```

```
0990            *=$0
```

```
0000    ;----------------------------------------------------------
0000    ; 6509 used to extend memory on bc2 and p2 systems
0000    ;   bits 0-5 used to direct:
0000    ;     execution register (4 bits)
0000    ;     indirect register (4 bits)
0000    ;
0000    ;   these bits can be expanded to sixteen (16) segment
0000    ;   control lines.  on 6509 reset all lines are high.
0000    ;
0000    ; current memory map:
0000    ; segment 15- $ffff-$e000  ram (kernal)
0000    ;             $dfff-$df00  i/o 6525 tpi2
0000    ;             $deff-$de00  i/o 6525 tpi1
0000    ;             $ddff-$dd00  i/o 6551 acia
0000    ;             $dcff-$dc00  i/o 6526 cia
0000    ;             $dbff-$db00  i/o unused (z80,8088,6809)
0000    ;             $daff-$da00  i/o 6581 sid
0000    ;             $d9ff-$d900  i/o unused (disks)
0000    ;             $d8ff-$d800  i/o 6566 vic/ 6845 80-col
0000    ;             $d7ff-$d400  colour nybles/80-col screen
0000    ;             $d3ff-$d000  video matrix/80-col screen
0000    ;             $cfff-$c000  character dot rom (p2 only)
0000    ;             $bfff-$8000  roms external (language)
0000    ;             $7fff-$4000  roms external (extensions)
0000    ;             $3fff-$2000  rom external
0000    ;             $1fff-$1000  rom internal
0000    ;             $0fff-$0400  unused
0000    ;             $03ff-$0002  ram (kernal/basic system)
0000    ; segment 14- segment 8 open (future expansion)
0000    ; segment 7 - $ffff-$0002  ram expansion (external)
0000    ; segment 6 - $ffff-$0002  ram expansion (external)
0000    ; segment 5 - $ffff-$0002  ram expansion (external)
0000    ; segment 4 - $ffff-$0002  ram expansion (external)
0000    ; segment 3 - $ffff-$0002  ram expansion
0000    ; segment 2 - $ffff-$0002  ram expansion
0000    ; segment 1 - $ffff-$0002  ram expansion
0000    ; segment 0 - $ffff-$0002  ram user/basic system
0000    ;
0000    ; the 6509 registers appear in locations $0000 and
0000    ; $0001 in all segments of memory.
0000    ;
0000    ;----------------------------------------------------------
0000    e6509    *=*+1        ;6509 execution register
0001    i6509    *=*+1        ;6509 indirection register
0002    irom     =$f          ;indirect=rom or execution=rom
0002             *=$90
0090    ;kernal page zero variables
0090    ;
0090    ;kernal indirect address variables
0090    ;
0090    fnadr    *=*+3        ;address of file name string
0093    sal      *=*+1        ;current load/store address
0094    sah      *=*+1
0095    sas      *=*+1
0096    eal      *=*+1        ;end of load/save
0097    eah      *=*+1
0098    eas      *=*+1
```

```
0099    stal      *=*+1        ;start of load/save
009a    stah      *=*+1
009b    stas      *=*+1
009c    ;
009c    ;frequently used kernal variables
009c    ;
009c    staus     *=*+1        ;i/o operation status
009d    fnlen     *=*+1        ;file name length
009e    la        *=*+1        ;current logical index
009f    fa        *=*+1        ;current first address
00a0    sa        *=*+1        ;current second address
00a1    dfltn     *=*+1        ;default input device
00a2    dflto     *=*+1        ;default output device
00a3    ;
00a3    ;tape buffer pointer
00a3    ;
00a3    tapel     *=*+3        ;address of tape buffer
00a6    ;
00a6    ; see kernal listing for allocation information
00a6    ;
00a6    ;screen editor page zero variables
00a6    ;
00a6    ;editor indirect address variables
00a6    ;
00a6              *=$c0        ;leave some space
00c0    pkybuf    *=*+2        ;start adr of pgm key
00c2    keypnt    *=*+2        ;current pgm key buf
00c4    sedsal    *=*+2        ;scroll ptr
00c6    sedeal    *=*+2        ;scroll ptr
00c8    pnt       *=*+2        ;current character pointer
00ca    ;
00ca    ;editor variables for speed and size
00ca    ;
00ca    tblx      *=*+1        ;cursor line
00cb    pntr      *=*+1        ;cursor column
00cc    grmode    *=*+1        ;graphic/text mode flag
00cd    lstx      *=*+1        ;last character index
00ce    lstp      *=*+1        ;screen edit start position
00cf    lsxp      *=*+1
00d0    crsw      *=*+1        ;
00d1    ndx       *=*+1        ;index to keyd queue
00d2    qtsw      *=*+1        ;quote mode flag
00d3    insrt     *=*+1        ;border colour
00d4    config    *=*+1        ;cursor type
00d5    indx      *=*+1        ;last byte position of line
00d6    kyndx     *=*+1        ;count program key string
00d7    rptcnt    *=*+1        ;delay tween chars
00d8    delay     *=*+1        ;delay to next repeat
00d9    ;
00d9    sedt1     *=*+1        ;frequently used temp variables
00da    sedt2     *=*+1
00db    ;
00db    ;frequently used editor variables
00db    ;
00db    data      *=*+1        ;current print data
00dc    sctop     *=*+1        ;top screen 0-25
00dd    scbot     *=*+1        ;bottom 0-25
```

```
00de    sclf     *=*+1        ;left margin
00df    scrt     *=*+1        ;right margin
00e0    modkey   *=*+1        ;keyscanner mode byte ($ff - no key down
                               last scan)
00e1    norkey   *=*+1        ;keyscanner normal byte ($ff - no key down
                               last scan)
00e2    bitabl   *=*+4        ;wrap bitmap
00e6    zpend
00e6    ;
00e6             *=$100
0100    ; stack space
0100             *=$200
0200    buf      *=*+256      ;basic's line input
0300    ;
0300    ; this area reserved for kernal absolutes
0300    ;  see kernal listing for other locations
0300    ;
0300    ctlvec =$0322
0300    escvec =$0320
0300    hiadr  =$0355
0300    bsout  =$ffd2         ;kernal vector
0300    ;
0300    ;screen editor absolute
0300    ;
0300             *=$380       ;block some area for editor
0380    pkyend   *=*+2        ;program key buffer end address
0382    keyseg   *=*+1        ;segment number for function key ram page
0383    keysiz   *=*+20       ;function key sizes ...don't clear...
0397    rvs      *=*+1        ;reverse field flag
0398    lintmp   *=*+1        ;line # tween in and out
0399    lstchr   *=*+1        ;last char printed
039a    insflg   *=*+1        ;auto insert flag
039b    scrdis   *=*+1        ;scroll disable flag
039c    fktmp                 ;also used for function key temporary
039c    bitmsk   *=*+1        ;temporary bitmask
039d    keyidx   *=*+1        ;index to programmables
039e    logscr   *=*+1        ;logical/physical scroll flag
039f    bellmd   *=*+1        ;flag to turn on end of line bell
03a0    pagsav   *=*+1        ;temp ram page
03a3    ;
03a3    tab      *=*+10       ;tabstop flags (80 max)
03ad    ;
03ad    keyd     *=*+10       ;key character queue
03b7    funvec   *=*+2        ;indirect jump vector for function keys
03b9    sedt3    *=*+1        ;another temp used during function key
                               listing
03ba    absend
03ba    ;
03ba    ; system warm start variables and vectors
03ba    ;
03ba             *=$3f8
03f8    evect    *=*+5
03fd    warm     =$a5         ;warm start flag
03fd    winit    =$5a         ;initialization complete flag
03fd             *=$400
0400    ramloc
0400             .end
```

```
0400               .lib basic-define
0400    ;           page zero storage definitions
0400               *=0
0000    ;
0000    e6509      *=*+1          ;execution bank
0001    i6509      *=*+1          ;indirection bank
0002    ;
0002    usrpok     *=*+3          ;set up origin by init
0005    tmhour     *=*+1          ;for ti$ calculations
0006    tmmin      *=*+1
0007    tmsec      *=*+1
0008    tmten      *=*+1
0009    ;
0009    form       *=*+ptrsiz     ;format pointer
000c    ;
000c    integr                    ;one-byte integer from qint
000c    charac     *=*+1          ;a delimiting char
000d    endchr     *=*+1          ;other delimiting char
000e    count      *=*+1          ;general counter
000f    xcnt       *=*+1          ;dos loop counter
0010    ;           flags
0010    ;           dimflg, valtyp and intflg must be
0010    ;           consecutive locations.
0010    dimflg     *=*+1          ;getting a pointer to a variable
0011    ;           it is important to remember whether
0011    ;           it is being done for dim or not.
0011    valtyp     *=*+1          ;the type indicator 0=numeric, 1=string
0012    intflg     *=*+1          ;tells if integer
0013    ;
0013    garbfl                    ;whether to do garbage collection
0013    dores      *=*+1          ;whether can or can't crunch res'd word.
0014    ;           turned on when data being scanned by
0014    ;           crunch so unquoted strs won't be crunched.
0014    ;
0014    subflg     *=*+1          ;flag whether sub'd variable allowed.
0015    ;           for and user-defined function pointer
0015    ;           fetching turn this on before calling
0015    ;           ptrget so arrays won't be detected.
0015    ;           stkini and ptrget clear it.
0015    ;           also disallows integers there.
0015    ;
0015    inpflg     *=*+1          ;flags are doing input or read.
0016    ;
0016    dsdesc     *=*+ptrsiz     ;disk status string
003e    channl     *=*+1          ;holds channel number
003f    poker
003f    linnum     *=*+2          ;location to store line #
0041    ;           pointers to temporary string descriptors.
0041    ;           temp descriptors are located in the string bank
0041    ;           hence, bank of strbnk is assumed for temppt,lastpt
0041    temppt     *=*+1          ;temppst relative offset to 1st free temp
                                   descr
0042    lastpt     *=*+2          ;pointer to last-used str temporary
0044    tempst     *=*+2          ;pointer to storage for 3 temporary
0046    ;                                                    descriptors.
0046    index
0046    index1     *=*+ptrsiz     ;direct cells for 1st indexing usage
```

```
0049    index2   *=*+ptrsiz   ;direct cells for 2nd indexing usage
004c    ;
004c    ;
004c    resho    *=*+1        ;result of multiplier and divider
004d    resmoh   *=*+1
004e    addend                ;temp used by umult
004e    resmo    *=*+1
004f    reslo    *=*+1
0050             *=*+1        ;overflow previous cells
0051    ;         pointers into dynamic data structures
0051    ;         all are 2-byte offsets into fixed banks
0051    ;         the following always mark the beginning of an area:
0051    ;                  txttab, vartab
0051    ;                  arytab, memtop
0051    ;         these will have unchangeable values in versions
0051    ;         where the areas they mark are equal to the "bottom"
0051    ;         (or "top" for memtop) of a bank.
0051    ;         additional variables:
0051    ;                  txtend, varend, aryend
0051    ;         are used to mark the end of an area, when the start
0051    ;         of the "next" area is in a different bank (i.e.,
0051    ;         the end isn't bordered by another area.)
0051    ;
0051    ;         highst is used to store the offset value from a basic
0051    ;         startup call to get the top of memory.
0051    ;
0051    ;         the limit of growth in an area must also be kept.
0051    ;         in the different versions, the following are used:
0051    ;
0051    ;         64k:        memtop (all)
0051    ;         128k        buffpt (text)
0051    ;                     memtop (data)
0051    ;         192k        buffpt (text)
0051    ;                     highst (arrays)
0051    ;                     memtop (vars,strs)
0051    ;         256k        buffpt (text)
0051    ;                     highst (vars)
0051    ;                     highst (arrays)
0051    ;                     memtop (strs)
0051    ;
0051    txttab   *=*+2        ;pointer to beginning of text and
0053    ;         doesn't change after being setup
0053    ;         by init
0053    ;
0053    txtend   *=*+2        ;pointer to end of text (except 64k)
0055    ;
0055    vartab   *=*+2        ;pointer to start off simple variable
0057    ;                                                   space.
0057    varend   *=*+2        ;pointer to end of simple vars (256k only)
0059    ;
0059    arytab   *=*+2        ;pointer to start of array table
005b    ;
005b    aryend   *=*+2        ;pointer end of arrays (192k, 256k only)
005d    ;
005d    strend   *=*+2        ;end of storage in use.
005f    ;
005f    fretop   *=*+2        ;top of str free space
```

```
0061    frespc   *=*+2         ;pointer to new str
0063    memtop   *=*+ptrsiz    ;highest location in memory
0066              line numbers and textual pointers
0066    curlin   *=*+2         ;current line number
0068    oldlin   *=*+2         ;old line number (setup by stop or
006a    ;         end in a program)
006a    oldtxt   *=*+ptrsiz    ;old text pointer
006d    ;
006d    datlin   *=*+2         ;data line number
006f    datptr   *=*+2         ;pointer to data. initialized to point
0071    ;         at the zero infront of (txttab) by
0071    ;         clr command.
0071    ;         updated by execution of a read.
0071    inpptr   *=*+2         ;remember where input is coming from.
0073    ;         stuff used in evaluations
0073    ;
0073    varnam   *=*+2         ;variable's name
0075    ;
0075    fdecpt                 ;pointer into power of tens table.
0075    varpnt   *=*+ptrsiz    ;pointer to variable in memory
0078    ;
0078    forpnt                 ;a variable's pointer for for loops
0078    ;         and let statements (3 bytes).
0078    lstpnt   *=*+ptrsiz    ;pointer to list string (3 bytes).
007b    ;
007b    vartxt                 ;save current txtptr on read.
007b    opptr    *=*+ptrsiz    ;pointer to current op's entry in optab.
007e    ;
007e    opmask   *=*+1         ;mask created by current operation.
007f    ;         temporary floating result registers (5 bytes each):
007f    ;              tempf1,tempf2,tempf3
007f    ;
007f    tempf3                 ;temp float reg
007f    grbpnt                 ;pointer used in garbage collection.
007f    defpnt   *=*+ptrsiz    ;pointer used in function definition.
0082    ;
0082    dscpnt   *=*+ptrsiz    ;pointer to a string descriptor.
0085    ;
0085    jmper    *=*+2         ;three bytes long
0087    oldov    *=*+1         ;the old overflow.
0088    ;
0088    tempf1                 ;temp float reg
0088    ptarg1=tempf1          ;multiply def'd for use by instr$
0088    ptarg2=tempf1+3
0088    str1=tempf1+6
0088    str2=tempf1+10
0088    tmppos=tempf1+14
0088    positn=tempf1+15
0088    match=tempf1+16
0088    arypnt                 ;pointer used in array building.
0088    highds   *=*+ptrsiz    ;destination of highest element in bit.
008b    hightr   *=*+ptrsiz    ;source of highest element to move.
008e    ;
008e    tempf2                 ;temp float reg (5 bytes)
008e    lowds    *=*+1         ;location of last byte transferred (3
                                                             bytes).
008f    deccnt   *=*+1         ;number of places before decimal point.
```

```
0090    tenexp   *=*+1          ;base ten exponent
0091    ;
0091    grbtop                  ;pointer used in garbage collection. (3
                                                                bytes)
0091    lowtr    *=*+1          ;last thing to move in bit (3 bytes)
0092    dptflg   *=*+1          ;has a dpt been input
0093    expsgn   *=*+1          ;sign of exponent
0094    ;         the floating accumulator
0094    dsctmp   *=*+1          ;temporary descriptors are built here.
0095                            ;dsctmp overlaps up to facmoh.
0095    fac
0095    facexp   *=*+1
0096    facho    *=*+1          ;most significant byte of mantissa.
0097    facmoh   *=*+1
0098    indice                  ;used by qint.
0098    facmo    *=*+1
0099    faclo    *=*+1
009a    facsgn   *=*+1
009b    degree                  ;count used by polynomials.
009b    sgnflg   *=*+1
009c    bits     *=*+1          ;cell for shiftr to use.
009d    ;         the floating argument (unpacked)
009d    t1=*                    ;temporaries --uses fp buffer
009d    t2=t1+1
009d    t3=t1+2
009d    t4=t1+3
009d    ;
009d    argexp   *=*+1
009e    argho    *=*+1
009f    argmoh   *=*+1
00a0    argmo    *=*+1
00a1    arglo    *=*+1
00a2    argsgn   *=*+1
00a3    strngl
00a3    arisgn   *=*+1          ;a sign reflecting the result
00a4    facov    *=*+1          ;overflow byte of the fac
00a5             *=*+1
00a6    strng2                  ;- > to str or desc
00a6    polypt                  ;- > to polynomial coefficients
00a6    curtol                  ;absolute linear index is formed here
00a6    fbufpt   *=*+ptrsiz     ;- > into fbuffr used by fout
00a9    txtptr   *=*+ptrsiz     ;pointer to current term
00ac    buffpt   *=*+ptrsiz     ;input buffer
00af    ;
00af    noze                    ;using's leading zero counter
00af    parsts   *=*+1          ;dos std parser word
00b0    point                   ;using's pointer to decimal point
00b0    parstx   *=*+1          ;dos aux parser word
00b1    ;
00b1    seedpt   *=*+2
00b3    errnum   *=*+1
00b4    ;         string area available for copy.  this area is used
00b4    ;         by fout as a buffer and must have dosspc contiguous
00b4    ;         bytes.
00b4    ;
00b4    ;         in addition this area is used to stored temporaries
00b4    ;         used by the dos interface routines, note, declaration
```

```
ØØb4    ;          order of locations dosofl-dossa must be preserved.
ØØb4    ;
ØØb4               *=$2ØØ
Ø2ØØ    fbuffr
Ø2ØØ    vspbuf                   ;buffer used to interface with vsp
Ø2ØØ               *=*+16        ;reserve 16 bytes for filename 1
Ø21Ø    dosfl1     *=*+1         ;dos file name 1 length
Ø211    dosds1     *=*+1         ;dos disk drive 1
Ø212    dosfla     *=*+2         ;dos file name 1 address
Ø214    dosflb     *=*+1         ;dos file name 1 bank
Ø215    ;
Ø215    dosf2l     *=*+1         ;dos file name 2 length
Ø216    dosds2     *=*+1         ;dos disk drive 2
Ø217    dosf2a     *=*+2         ;dos file name 2 address
Ø219    dosf2b     *=*+1         ;dos file name 2 bank
Ø21a    ;
Ø21a    dosbnk     *=*+1         ;dos bank number
Ø21b    dosofl     *=*+2         ;dos low offset (bsave,bload)
Ø21d    dosofh     *=*+2         ;dos high offset (bsave)
Ø21f    ;
Ø21f    dosla      *=*+1         ;dos logical address
Ø22Ø    dosfa      *=*+1         ;dos physical address
Ø221    dossa      *=*+1         ;dos secondary address
Ø222    dosrcl     *=*+1         ;dos record length
Ø223    ;
Ø223    dosdid     *=*+2         ;dos disk identifier(2 chars)
Ø225    didchk     *=*+1         ;dos did flag
Ø226    ;
Ø226    dosstr     *=*+1         ;dos output string buffer
Ø227    dosspc=*-fbuffr          ;space used by dos routines
ØØ27               *=*+46
Ø255    ;
Ø255    ;
Ø255    trmpos                   ;cursor column on crt
Ø255    andmask    *=*+1         ;mask used by wait
Ø256    eormsk     *=*+1         ;mask used by wait
Ø257    ;
Ø257    dfbank     *=*+1         ;default bank number
Ø258    dolu       *=*+1         ;default output lu (Ø=> not std output.)
Ø259    ;                                              keeps ds + dir ok
Ø259    domask
Ø259    tansgn     *=*+1          ;used in determining sign of tan
Ø25a    ;
Ø25a    ldaabs     *=*+1         ;lda abs routine (see initat)
Ø25b    tttemp                   ;temporary store
Ø25b    ldaadr     *=*+2         ;modifiable address
Ø25d               *=*+1         ;return opcode
Ø25e    ;
Ø25e    ;declarations for print using
Ø25e    ;
Ø25e    hulp       *=*+1         ;counter
Ø25f    bnr        *=*+1         ;pointer to begin no
Ø26Ø    enr        *=*+1         ;pointer to end no
Ø261    dolr       *=*+1         ;dollar flag
Ø262    flag       *=*+1         ;comma flag
Ø263    swe        *=*+1         ;counter
Ø264    usgn       *=*+1         ;sign exponent
```

```
0265    uexp      *=*+1        ;pointer to exponent
0266    vn        *=*+1        ;# digits before decimal point
0267    chsn      *=*+1        ;justify flag
0268    vf        *=*+1        ;# pos before dec point (field)
0269    nf        *=*+1        ;# pos after dec point (field)
026a    posp      *=*+1        ;+/- flag (field)
026b    fesp      *=*+1        ;exponent flag (field)
026c    etof      *=*+1        ;switch
026d    cform     *=*+1        ;char counter (field)
026e    sno       *=*+1        ;sign no
026f    blfd      *=*+1        ;blank/star flag
0270    begfd     *=*+1        ;pointer to begin of field
0271    lfor      *=*+1        ;length of for at
0272    endfd     *=*+1        ;pointer to end of field
0273    puchrs
0273    pufill    *=*+1        ;print using fill symbol
0274    pucoma    *=*+1        ;print using comma symbol
0275    pudot     *=*+1        ;print using decimal point symbol
0276    pumony    *=*+1        ;print using monetary symbol
0277              *=$280
0280    ;          basic indirects
0280    ;
0280    ierror    *=*+2        ;error routine, output err in .x
0282    imain     *=*+2        ;interpreter main loop
0284    icrnch    *=*+2        ;tokenization routine
0286    iqplop    *=*+2        ;token output expander routine
0288    igone     *=*+2        ;dispatcher
028a    ieval     *=*+2        ;eval routine
028c    ifrmev    *=*+2        ;frmevl routine
028e    ichrgo    *=*+2        ;chrgot routine
0290    ichrge    *=*+2        ;chrget routine
0292    adray1    *=*+2        ;convert float -> integer
0294    adray2    *=*+2        ;convert integer -> float
0296    ;          error trapping declarations
0296    ;
0296    trapno    *=*+2        ;error trap vector
0298    errlin    *=*+2        ;holds line # of last error
029a    errtxt    *=*+2        ;text pointer at time of error
029c    oldstk    *=*+1        ;stack pointer before execution of last
                                instruction
029d    tmptrp    *=*+1        ;used to save hi byte of trap line >trap
                                &  resume
029e    dsptmp    *=*+1        ;temporary for dispose
029f    oldtok    *=*+1        ;
02a0    tmpdes    *=*+6        ;temporary for instr$
02a6    ;
02a6    highst    *=*+2        ;max offset for any user bank
02a8                           ;
02a8                           ;
02a8    msiism    *=*+1        ;used to save length of string to be added
02a9    newsys=$ff6c                                 in garb collect
02a9                           ;
02a9    .end
02a9    .lib kernal-equate
02a9    ;tape block types
02a9    ;
02a9    eot        = 5         ;end of tape
```

```
02a9    blf       = 1         ;basic load file
02a9    bdf       = 2         ;basic data file
02a9    bdfh      = 4         ;basic data file header
02a9    bufsiz    = 192       ;buffer size
02a9    cr        = $d        ;carriage return
02a9    basic     = $8000     ;start of rom (language)
02a9    kernal    = $e000     ;start of rom (kernal)
02a9    ;  6845 video display controller for bc2
02a9    ;
02a9    vdc       = $d800
02a9    adreg     = $0        ;address register
02a9    dareg     = $1        ;data register
02a9    ;  6581 sid sound interface device
02a9    ;    register list
02a9    sid       = $da00
02a9    ;
02a9    ; base addresses osc1, osc2, osc3
02a9    osc1      = $00
02a9    osc2      = $07
02a9    osc3      = $0e
02a9    ;
02a9    ; osc registers
02a9    freqlo    = $00
02a9    freqhi    = $01
02a9    pulsef    = $02
02a9    pulsec    = $03
02a9    oscctl    = $04
02a9    atkdcy    = $05
02a9    susrel    = $06
02a9    ;
02a9    ; filter ocntrol
02a9    fclow     = $15
02a9    fchi      = $16
02a9    resnce    = $17
02a9    volume    = $18
02a9    ;
02a9    ; pots, random number, and env3 out
02a9    potx      = $19
02a9    poty      = $1a
02a9    random    = $1b
02a9    env3      = $1c
02a9    ; 6526 cia complex interface adapter
02a9    ;  game / ieee data / user
02a9    ;
02a9    ;   timer a: ieee local / cass local / music / game
02a9    ;   timer b: ieee deadm / cass deadm / music / game
02a9    ;
02a9    ;   pra0 : ieee data1 / user / paddle game 1
02a9    ;   pra1 : ieee data2 / user / paddle game 2
02a9    ;   pra2 : ieee data3 / user
02a9    ;   pra3 : ieee data4 / user
02a9    ;   pra4 : ieee data5 / user
02a9    ;   pra5 : ieee data6 / user
02a9    ;   pra6 : ieee data7 / user / game trigger 14
02a9    ;   pra7 : ieee data8 / user / game trigger 24
02a9    ;
02a9    ;   prb0 : user / game 10
```

```
02a9    ;   prb1 : user / game 11
02a9    ;   prb2 : user / game 12
02a9    ;   prb3 : user / game 13
02a9    ;   prb4 : user / game 20
02a9    ;   prb5 : user / game 21
02a9    ;   prb6 : user / game 22
02a9    ;   prb7 : user / game 23
02a9    ;
02a9    ;   flag : user / cassette read
02a9    ;   pc   : user
02a9    ;   ct   : user
02a9    ;   sp   : user
02a9    ;
02a9    cia       = $dc00
02a9    pra       = $0         ;data reg a
02a9    prb       = $1         ;data reg b
02a9    ddra      = $2         ;direction reg a
02a9    ddrb      = $3         ;direction reg b
02a9    talo      = $4         ;timer a low byte
02a9    tahi      = $5         ;timer a high byte
02a9    tblo      = $6         ;timer b low byte
02a9    tbhi      = $7         ;timer b high byte
02a9    tod10     = $8         ;10ths of seconds
02a9    todsec    = $9         ;seconds
02a9    todmin    = $a         ;minutes
02a9    todhr     = $b         ;hours
02a9    sdr       = $c         ;serial data register
02a9    icr       = $d         ;interrupt control register
02a9    cra       = $e         ;control register a
02a9    crb       = $f         ;control register b
02a9    ;
02a9    ; 6526 cia for inter-process communication
02a9    ;
02a9    ;   pra   = data port
02a9    ;   prb0  = busy1 (1=>6509 off dbus)
02a9    ;   prb1  = busy2 (1=>8088/z80 off dbus)
02a9    ;   pra2  = semaphore 8088/z80
02a9    ;   prb3  = semaphore 6509
02a9    ;   prb4  = unused
02a9    ;   prb5  = unused
02a9    ;   prb6  = irq to 8088/z80 (lo)
02a9    ;   prb7  = unused
02a9    ;
02a9    ipcia     = $db00
02a9    ;
02a9    sem88     = $04        ;prb bit2
02a9    sem65     = $08        ;prb bit3
02a9    ; 6551 acia rs232c and network interface
02a9    ;
02a9    acia      = $dd00
02a9    drsn      = $00        ;transmit/receive data register
02a9    srsn      = $01        ;status register
02a9    cdr       = $02        ;command register
02a9    ctr       = $03        ;control register
02a9    dsrerr    = $40        ;data set ready error
02a9    dcderr    = $20        ;data carrier detect error
02a9    doverr    = $08        ;receiver outer buffer overrun
```

```
02a9    ; 6525 tpil triport interface device #1
02a9    ;  ieee control / cassette / network / vic / irq
02a9    ;
02a9    ;   pa0  : ieee dc control (ti parts)
02a9    ;   pa1  : ieee te control (ti parts) (t/r)
02a9    ;   pa2  : ieee ren
02a9    ;   pa3  : ieee atn
02a9    ;   pa4  : ieee day
02a9    ;   pa5  : ieee eoi
02a9    ;   pa6  : ieee ndac
02a9    ;   pa7  : ieee nrfd
02a9    ;
02a9    ;   pb0  : ieee ifc
02a9    ;   pb1  : ieee srq
02a9    ;   pb2  : network transmitter enable
02a9    ;   pb3  : network receiver enable
02a9    ;   pb4  : arbitration logic switch
02a9    ;   pb5  : cassette write
02a9    ;   pb6  : cassette motor
02a9    ;   pb7  : cassette switch
02a9    ;
02a9    ;   irq0  : 50/60 hz irq
02a9    ;   irq1 : ieee srq
02a9    ;   irq2 : 6526 irq
02a9    ;   irq3 : (opt) 6526 inter-processor
02a9    ;   irq4 : 6551
02a9    ;   *irq : 6566 (vic) / user devices
02a9    ;   cb   : vic dot select
02a9    ;   ca   : vic matrix select
02a9    ;
02a9    tpil      = $de00
02a9    pa        = $0        ;port register a
02a9    pb        = $1        ;port register b
02a9    pc        = $2        ;port register c
02a9    lir       = $2        ;interrupt latch register mc=1
02a9    ddpa      = $3        ;data direction register a
02a9    ddpb      = $4        ;data direction register b
02a9    ddpc      = $5        ;data direction register c
02a9    mir       = $5        ;interrupt mask register mc=1
02a9    creg      = $6        ;control register
02a9    air       = $7        ;active interrupt register
02a9    ;
02a9    freq      = $01       ;irq line 50/60 hz found on ...
02a9    ; 6525 tpi2 tirport interface device #2
02a9    ;  keyboard / vic 16k control
02a9    ;
02a9    ;   pa0  : kybd out 8
02a9    ;   pa1  : kybd out 9
02a9    ;   pa2  : kybd out 10
02a9    ;   pa3  : kybd out 11
02a9    ;   pa4  : kybd out 12
02a9    ;   pa5  : kybd out 13
02a9    ;   pa6  : kybd out 14
02a9    ;   pa7  : kybd out 15
02a9    ;
02a9    ;   pb0  : kybd out 0
02a9    ;   pb1  : kybd out 1
```

```
02a9    ;    pb2  : kybd out 2
02a9    ;    pb3  : kybd out 3
02a9    ;    pb4  : kybd out 4
02a9    ;    pb5  : kybd out 5
02a9    ;    pb6  : kybd out 6
02a9    ;    pb7  : kybd out 7
02a9    ;
02a9    ;    pc0  : kybd in 0
02a9    ;    pc1  : kybd in 1
02a9    ;    pc2  : kybd in 2
02a9    ;    pc3  : kybd in 3
02a9    ;    pc4  : kybd in 4
02a9    ;    pc5  : kybd in 5
02a9    ;    pc6  : vic 16k bank select low
02a9    ;    pc7  : vic 16k bank select hi
02a9    ;
02a9    tpi2       = $df00
02a9    ; ieee line equates
02a9    ;
02a9    dc         = $01       ;75160/75161 control line
02a9    te         = $02       ;75160/75161 control line
02a9    ren        = $04       ;remote enable
02a9    atn        = $08       ;attention
02a9    dav        = $10       ;data available
02a9    eoi        = $20       ;end or identify
02a9    ndac       = $40       ;not data accepted
02a9    nrfd       = $80       ;not ready for data
02a9    ifc        = $01       ;interface clear
02a9    srq        = $02       ;service request
02a9    ;
02a9    rddb       = nrfd+ndac+te+dc+ren    ;directions for receiver
02a9    tddb       = eoi+dav+atn+te+dc+ren  ;directions for transmit
02a9    ;
02a9    eoist      = $40       ;eoi status test
02a9    tlkr       = $40       ;device is talker
02a9    listnr     = $20       ;device is listener
02a9    utlkr      = $5f       ;device untalk
02a9    ulstn      = $3f       ;device unlisten
02a9    ;
02a9    toout      = $01       ;timeout status on output
02a9    toin       = $02       ;timeout status on input
02a9    eoist      = $40       ;eoi on input
02a9    nodev      = $80       ;no device on bus
02a9    sperr      = $10       ;verify error
02a9    ;
02a9    ; equates for c3p0 flag bits 6 and 7
02a9    ;
02a9    slock      = $40       ;screen editor lock-out
02a9    dibf       = $80       ;data in output buffer
02a9               .end
02a9               .lib scrn-equate
02a9    ;tape block types
02a9    ;
02a9    eot        = 5         ;end of tape
02a9    blf        = 1         ;basic load file
02a9    bdf        = 2         ;basic data file
02a9    bdfh       = 4         ;basic data file header
```

```
02a9    bufsiz    = 192        ;buffer size
02a9    cr        = $d         ;carriage return
02a9    basic     = $8000      ;start of rom (language)
02a9    kernal    = $e000      ;start of rom (kernal)
02a9    ; 6845 video display controller for bc2
02a9    ;
02a9    vdc       = $d800
02a9    adreg     = $0         ;address register
02a9    dareg     = $1         ;data register
02a9    ; 6581 sid sound interface device
02a9    ;   register list
02a9    sid       = $da00
02a9    ;
02a9    ; base addresses osc1, osc2, osc3
02a9    osc1      = $00
02a9    osc2      = $07
02a9    osc3      = $0e
02a9    ;
02a9    ; osc registers
02a9    freqlo    = $00
02a9    freqhi    = $01
02a9    pulsef    = $02
02a9    pulsec    = $03
02a9    oscctl    = $04
02a9    atkdcy    = $05
02a9    susrel    = $06
02a9    ;
02a9    ; filter control
02a9    fclow     = $15
02a9    fchi      = $16
02a9    resnce    = $17
02a9    volume    = $18
02a9    ;
02a9    ; pots, random number, and env3 out
02a9    potx      = $19
02a9    poty      = $1a
02a9    random    = $1b
02a9    env3      = $1c
02a9    ; 6526 cia complex interface adapter
02a9    ;  game / ieee data / user
02a9    ;
02a9    ;    timer a: ieee local / cass local / music / game
02a9    ;    timer b: ieee deadm / cass deadm / music / game
02a9    ;
02a9    ;    pra0 : ieee data1 / user
02a9    ;    pra1 : ieee data2 / user
02a9    ;    pra2 : ieee data3 / user
02a9    ;    pra3 : ieee data4 / user
02a9    ;    pra4 : ieee data5 / user
02a9    ;    pra5 : ieee data6 / user
02a9    ;    pra6 : ieee data7 / user / game trigger 14
02a9    ;    pra7 : ieee data8 / user / game trigger 24
02a9    ;
02a9    ;    prb0 : user / game 10
02a9    ;    prb1 : user / game 11
02a9    ;    prb2 : user / game 12
02a9    ;    prb3 : user / game 13
```

```
02a9    ;   prb4 : user / game 20
02a9    ;   prb5 : user / game 21
02a9    ;   prb6 : user / game 22
02a9    ;   prb7 : user / game 23
02a9    ;
02a9    ;   flag : user
02a9    ;   pc   : user
02a9    ;   ct   : user
02a9    ;   sp   : user
02a9    ;
02a9    cia       = $dc00
02a9    pra       = $0        ;data reg a
02a9    prb       = $1        ;data reg b
02a9    ddra      = $2        ;direction reg a
02a9    ddrb      = $3        ;direction reg b
02a9    talo      = $4        ;timer a low byte
02a9    tahi      = $5        ;timer a high byte
02a9    tblo      = $6        ;timer b low byte
02a9    tbhi      = $7        ;timer b high byte
02a9    tod10     = $8        ;10ths of seconds
02a9    todsec    = $9        ;seconds
02a9    todmin    = $a        ;minutes
02a9    todhr     = $b        ;hours
02a9    sdr       = $c        ;serial data register
02a9    icr       = $d        ;interrupt control register
02a9    cra       = $e        ;control register a
02a9    crb       = $f        ;control register b
02a9    ; 6551 acia rs232c and network interface
02a9    ;
02a9    acia      = $dd00
02a9    drsn      = $00       ;transmit/receive data register
02a9    srsn      = $01       ;status register
02a9    cdr       = $02       ;command register
02a9    ctr       = $03       ;control register
02a9    dsrerr    = $40       ;data set ready error
02a9    dcderr    = $20       ;data carrier detect error
02a9    doverr    = $08       ;receiver outer buffer overrun
02a9    ; 6525 tpil triport interface device #1
02a9    ;  ieee control / cassette / network / vic / irq
02a9    ;
02a9    ;   pa0  : ieee dc control (ti parts)
02a9    ;   pa1  : ieee tc control (ti parts) (t/r)
02a9    ;   pa2  : ieee ren
02a9    ;   pa3  : ieee atn
02a9    ;   pa4  : ieee dav
02a9    ;   pa5  : ieee eoi
02a9    ;   pa6  : ieee ndac
02a9    ;   pa7  : ieee nrfd
02a9    ;
02a9    ;   pb0  : ieee ifc
02a9    ;   pb1  : ieee srq
02a9    ;   pb2  : network transmitter enable
02a9    ;   pb3  : network receiver enable
02a9    ;   pb4  : arbitration logic switch
02a9    ;   pb5  : cassette write
02a9    ;   pb6  : cassetter motor
02a9    ;   pb7  : cassette switch
```

```
02a9    ;
02a9    ;   irq0 : 50/60 hz irq
02a9    ;   irq1 : ieee srq
02a9    ;   irq2 : 6526 irq
02a9    ;   irq3 : cassette read
02a9    ;   irq4 : 6551
02a9    ;   *irq : 6566 (vic) / user devices
02a9    ;   cb   : vic dot select
02a9    ;   ca   : vic matrix select
02a9    ;
02a9    tpi1      = $de00
02a9    pa        = $0        ;port register a
02a9    pb        = $1        ;port register b
02a9    pc        = $2        ;port register c
02a9    lir       = $2        ;interrupt latch register mc=1
02a9    ddpa      = $3        ;data direction register a
02a9    ddpb      = $4        ;data direction register b
02a9    ddpc      = $5        ;data direction register c
02a9    mir       = $5        ;interrupt mask register mc=1
02a9    creg      = $6        ;control register
02a9    air       = $7        ;active interrupt register
02a9    ;
02a9    freq      = $01       ;irq line 50/60 hz found on...
02a9    id55hz    = 27        ;55 hz value required by ioinit
02a9    ; 6525 tpi2 tirport interface device #2
02a9    ;  keyboard / vic 16k control
02a9    ;
02a9    ;   pa0  : kybd out 8
02a9    ;   pa1  : kybd out 9
02a9    ;   pa2  : kybd out 10
02a9    ;   pa3  : kybd out 11
02a9    ;   pa4  : kybd out 12
02a9    ;   pa5  : kybd out 13
02a9    ;   pa6  : kybd out 14
02a9    ;   pa7  : kybd out 15
02a9    ;
02a9    ;   pb0  : kybd out 0
02a9    ;   pb1  : kybd out 1
02a9    ;   pb2  : kybd out 2
02a9    ;   pb3  : kybd out 3
02a9    ;   pb4  : kybd out 4
02a9    ;   pb5  : kybd out 5
02a9    ;   pb6  : select for monitor(hish=ntsc,low=pal)
02a9    ;   pb0  : select for head(high=built-in,low=monitor)
02a9    ;
02a9    tpi2      = $df00
02a9    ; ieee line equates
02a9    ;
02a9    dc        = $01       ;75160/75161 control line
02a9    te        = $02       ;75160/75161 control line
02a9    ren       = $04       ;remote enable
02a9    atn       = $08       ;attention
02a9    dav       = $10       ;data available
02a9    eoi       = $20       ;end or identify
02a9    ndac      = $40       ;not data accepted
02a9    ndfd      = $80       ;not ready for data
02a9    ifc       = $01       ;interface clear
```

```
02a9    srq       = $02         ;service request
02a9    ;
02a9    rddb      = nrfd+ndac+te+dc    ;directions for receiver
02a9    tddb      = eoi+dav+atn+te+dc  ;directions for transmit
02a9    ;
02a9    eoist     = $40         ;eoi status test
02a9    tlkrt     = $40         ;device is talker
02a9    lstnr     = $20         ;device is listener
02a9    utlkr     = $5f         ;device untalk
02a9    ulstn     = $3f         ;device unlisten
02a9    ;
02a9    toout     = $01         ;timeout status on output
02a9    toin      = $02         ;timeout status on input
02a9    eoist     = $40         ;eoi on input
02a9    nodev     = $80         ;no device on bus
02a9    sperr     = $10         ;verify error
02a9    ;
02a9    ;         equates for c3p0 flag bits 6 and 7
02a9    ;
02a9    slock     = $40         ;screen editor lock-out
02a9    dibf      = $80         ;data in output buffer
02a9              .end
02a9              .end
```

IEEE Connector

Pin	ID	IC	Use	Address	
1	D1	CIA 6526	PRA 0	dc00	55320
2	D2	CIA 6526	PRA 1	dc00	56320
3	D3	CIA 6526	PRA 2	dc00	56320
4	D4	CIA 6526	PRA 3	dc00	56320
5	EOI	TPI 6525	PRA 5	de00	56832
6	DAV	TPI 6525	PRA 4	de00	56832
7	NRFD	TPI 6525	PRA 7	de00	56832
8	NDAC	TPI 6525	PRA 6	de00	56832
9	IFC	TPI 6525	PRB 0	de01	56833
10	SRQ	TPI 6525	PRB 1	de01	56833
11	ATN	TPI 6525	PRA 3	de00	56832
12	SHIELD				
A	D5	CIA 6526	PRA 4	dc00	56320
B	D6	CIA 6526	PRA 5	dc00	56320
C	D7	CIA 6526	PRA 6	dc00	56320
D	D8	CIA 6526	PRA 7	dc00	56320
E	REN	TPI 6525	PRA 2	de00	56832
F	GND				
H	GND				
J	GND				
K	GND				
L	GND				
M	GND				
N	GND				

RS232 Connector

Pin	ID
1	SHIELD
2	TxD
3	RxD
4	RTS
5	CTS
6	DSR
7	GND
8	DCD
9	N.C.
10	N.C.
11	+ 5 V DC
12	- 12 V DC
13	N.C.
14	N.C.
15	N.C.
16	N.C.
17	N.C.
18	N.C.
19	N.C.
20	DTR
21	N.C.
22	N.C.
23	N.C.
24	RxC
25	N.C.

USER Connector (internal)

Pin	ID	IC	Use	address	
1	GND				
2	PB2	TPI 6525	PRB 2	de01	56833
3	GND				
4	PB3	TPI 6525	PRB 3	de01	56833
5	NOT PC	CIA 6526	-PC		
	(Handshake PRB I/O, Output)				
6	NOT FL.	Cass-Read	-FLAG		
	(Interrupt, Input)				
7	2D7	CIA 6526	PRB 7	dc01	56321
8	2D6	CIA 6526	PRB 6	dc01	56321
9	2D5	CIA 6526	PRB 5	dc01	56321
10	2D4	CIA 6526	PRB 4	dc01	56321
11	2D3	CIA 6526	PRB 3	dc01	56321
12	2D2	CIA 6526	PRB 2	dc01	56321
13	2D1	CIA 6526	PRB 1	dc01	56321
14	2D0	CIA 6526	PRB 0	dc01	56321
15	1D7	CIA 6526	PRA 7	dc00	56320
16	1D6	CIA 6526	PRA 6	dc00	56320
17	1D5	CIA 6526	PRA 5	dc00	56320
18	1D4	CIA 6526	PRA 4	dc00	56320
19	1D3	CIA 6526	PRA 3	dc00	56320
20	1D2	CIA 6526	PRA 2	dc00	56320
21	1D1	CIA 6526	PRA 1	dc00	56320
22	1D0	CIA 6526	PRA 0	dc00	56320
23	NOT CNT	CIA 6526	-CNT	dc04/5	56324/5
24	+ 5 V DC				
25	NOT IRQ	TPI 6525	PRC 5	de02	56834
26	SP	CIA 6526	SP		
	(Serial Port I/O)				

Keyboard Connector (internal or external)

Pin	ID	IC	Use	address
1	PA0	TPI 6525	PRA 0	df00 57088
2	PA2	TPI 6525	PRA 2	df00 57088
3	PA4	TPI 6525	PRA 4	df00 57088
4	PA6	TPI 6525	PRA 6	df00 57088
5	PB0	TPI 6525	PRB 0	df01 57089
6	PB1	TPI 6525	PRB 1	df01 57089
7	PB2	TPI 6525	PRB 2	df01 57089
8	PB3	TPI 6525	PRB 3	df01 57089
9	PB4	TPI 6525	PRB 4	df01 57089
10	PB5	TPI 6525	PRB 5	df01 57089
11	PB6	TPI 6525	PRB 6	df01 57089
12	PB7	TPI 6525	PRB 7	df01 57089
13	PC5	TPI 6525	PRC 5	df02 57090
14	PA1	TPI 6525	PRA 1	df00 57088
15	PA3	TPI 6525	PRA 3	df00 57088
16	PA5	TPI 6525	PRA 5	df00 57088
17	PA7	TPI 6525	PRA 7	df00 57088
18	PC0	TPI 6525	PRC 0	df02 57090
19	PC1	TPI 6525	PRC 1	df02 57090
20	PC2	TPI 6525	PRC 2	df02 57090
21	PC3	TPI 6525	PRC 3	df02 57090
22	GND			
23	GND			
24	GND			
25	PC4	TPI 6525	PRC 4	df00 57090

Cartridge Connector

Pin	ID		
1	A0		
2	A1		
3	A2		
4	A3		
5	A4		
6	A5	Address	
7	A6		
8	A7		
9	A8		
10	A9		
11	A10		
12	A11		
13	A12		
14	+ 5 V DC		
15	+ 5 V DC		
A	BD0		
B	BD1		
C	BD2		
D	BD3	Data	
E	BD4		
F	BD5		
H	BD6		
J	BD7		
K	GND		
L	GND		
M	S R/W		
N	SO2		
P	NOT CSBANK 1	$2000-4000	
R	NOT CSBANK 2	$4000-6000	BANK 15
S	NOT CSBANK 3	$6000-8000	

Co-Processor Connector (internal)

Pin	ID
1	EXTMA3
2	DRAM00
3	EXTMA2
4	DRAM01
5	EXTMA7
6	DRAM02
7	EXTMA6
8	DRAM03
9	EXTMA5
10	DRAM04
11	EXTMA4
12	DRAM05
13	EXTMA1
14	DRAM06
15	EXTMA0
16	DRAM07
17	GND
18	GND
19	GND
20	GND
21	GND
22	NOT BUSY 1
23	GND
24	NOT P2REFREQ
25	GND
26	NOT P2REFGRNT
27	GND
28	BP0
29	GND
30	BP1
31	GND
32	BP2
33	N.C.
34	BP3
35	NOT PROCRES
36	NOT BUSY 2
37	EXTBUF R/W
38	NOT ERAS
39	DRAM R/W
40	NOT ECAS

Expansion Connector (internal)

Pin	ID	Use/address
1	+ 5 V DC	
2	+ 5 V DC	
3	+ 5 V DC	
4	+ 5 V DC	
5	GND	
6	GND	
7	GND	
8	GND	
9	GND	
10	GND	
11	NOT BRAS	DRAM: Row Access
12	IRQ3	PRC3 de02 56834
13	- 12 V DC	
14	NOT EXTRES	Reset
15	+ 12 V DC	
16	NOT S0	S0
17	NOT RES	System Reset
18	LPEN	Light Pen
19	S R/W	System Read/Write
20	NOT EXTBUFCS	Address: $0800-$0fff
21	TODCLK	50 Hz
22	NOT DISKROMCS	Address: $1000-$1fff
23	BDOTCLK	(18 MHz)
24	No Connection	
25	S02	phi 2
26	NOT BCAS	DRAM: Column Access
27	S01	phi 1
28	NOT CS1	Address: $d900-$d9ff
29	BD3	Data
30	NOT EXTPRTCS	Address: $db00-$dbff
31	BD4	Data
32	BD2	Data
33	BD5	Data
34	BD1	Data
35	BD7	Data
36	BD0	Data
37	BA13	Address
38	BD7	Data
39	BA14	Address
40	BA15	Address
41	BA1	Address
42	BA0	Address
43	BA2	Address
44	BA11	Address
45	BA3	Address
46	BA10	Address
47	BA12	Address
48	BA4	Address
49	BA9	Address
50	BA5	Address
51	BA8	Address
52	BA6	Address

53	BP0	Bank
54	BA7	Address
55	BP1	Bank
56	BP2	Bank
57	NOT NMI	Non-maskable Interrupt
58	BP3	Bank
59	RDY	Ready
60	NOT IRQ	Interrupt Request

Audio Connector

Pin	Use
1	Loudspeaker (8 ohm)
2	Not connected
3	Loudspeaker (8 ohm)

Power Connector

Pin	Use
1	50 Hz
2	- 12 V DC
3	+ 12 V DC
4	GND
5	GND
6	+ 5 V DC

Video Connector

Pin	Use
1	VIDEO
2	GND
3	VERTICAL SYNC
4	GND
5	HORIZONTAL SYNC
6	KEY
7	GND

APPENDIX G (For ASC and CHR$ Codes)

ASCII Code	Character/function
0	None (1)
1	CTRL-a (2)
2	CTRL-b or Commodore Key (2)
3	CTRL-c (2)
4	CTRL-d or CE (2)
5	CTRL-e (2)
6	CTRL-f (2)
7	CTRL-g or Bell (2)
8	CTRL-h (2)
9	CTRL-i or TAB (2)
10	CTRL-j (2)
11	CTRL-k (2)
12	CTRL-l (2)
13	CTRL-m or CTRL-RETURN or ENTER or CTRL-SHIFT-SPACE (2)
14	CTRL-n or NORM (2)
15	CTRL-o or Set Top (2)
16	CTRL-p (2)
17	CTRL-q or Cursor Down (2)
18	CTRL-r or RVS (2)
19	CTRL-s or Home (2)
20	CTRL-t or Delete (2)
21	CTRL-u (2)
22	CTRL-v (2)
23	CTRL-w (2)
24	CTRL-x (2)
25	CTRL-y or Cursor Up (2)
26	CTRL-z (2)
27	ESC or SHIFT ESC
28	RVS-pound
29	RVS-] or Cursor Right or SHIFT-Cursor Right
30	RVS-↑
31	RVS-back arrow
32	space
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7

56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@
65	a or A
66	b or B
67	c or C
68	d or D
69	e or E
70	f or F
71	g or G
72	h or H
73	i or I
74	j or J
75	k or K
76	l or L
77	m or M
78	n or N
79	o or O
80	p or P
81	q or Q
82	r or R
83	s or S
84	t or T
85	u or U
86	v or V
87	w or W
88	x or X
89	y or Y
90	z or Z
91	[
92	pound
93	]
94	↑
95	back arrow
96	SHIFT-space
97	!
98	"
99	#
100	$
101	%
102	&
103	'
104	(
105	)
106	*
107	+
108	,
109	-
110	.
111	/
112	0

113	1	
114	2	
115	3	
116	4	
117	5	
118	6	
119	7	
120	8	
121	9	
122	:	
123	;	
124	<	
125	=	
126	>	
127	?	
128	RVS-graphic	(2)
129	RVS-graphic or RVS-A	(2)
130	RVS-graphic or RVS-B or SHIFT-Commodore	(2)
131	SHIFT-RUN	(3)
132	RVS-graphic or RVS-D or SHIFT-CE	(2)
133	RVS-graphic or RVS-E	(2)
134	RVS-graphic or RVS-F	(2)
135	RVS-graphic or RVS-G	(2)
136	RVS-graphic or RVS-H	(2)
137	RVS-graphic or RVS-I or SHIFT-TAB	(2)
138	RVS-graphic or RVS-J	(2)
139	RVS-graphic or RVS-K	
140	RVS-graphic or RVS-L	
141	SHIFT-Return or SHIFT-Enter	(4)
142	RVS-graphic or RVS-N or GRAPH	(2)
143	RVS-graphic or RVS-O or Set Bottom	(2)
144	RVS-graphic or RVS-P	(2)
145	RVS-graphic or RVS-Q or Cursor Up	(2)
146	RVS-graphic or RVS-R or SHIFT-RVS	(2)
147	RVS-graphic or RVS-S or SHIFT-CLR	(2)
148	SHIFT-DEL	(5)
149	RVS-graphic or RVS-U	(2)
150	RVS-graphic or RVS-V	(2)
151	RVS-graphic or RVS-W	(2)
152	RVS-graphic or RVS-X	(2)
153	RVS-graphic or RVS-Y	(2)
154	RVS-graphic or RVS-Z	(2)
155	RVS-graphic	(2)
156	RVS-graphic	(6)
157	RVS-graphic or Cursor Left	(2)
158	RVS-pi	
159	RVS-graphic	(6)
160	SHIFT-space	
161	CTRL-1 or graphic	
162	CTRL-2 or graphic	
163	CTRL-3 or graphic	(6)
164	CTRL-4 or graphic	
165	CTRL-5 or graphic	
166	CTRL-pi or graphic	(6)
167	CTRL-6 or graphic	(6)
168	CTRL-' or graphic	(7)
169	CTRL-= or graphic	(8)

170	CTRL-- or graphic	(6)(9)
171	CTRL-+ or graphic	(6)(9)
172	CTRL-0 or graphic	
173	CTRL-0 or graphic	(9)
174	CTRL-2 or graphic	(9)
175	CTRL-/ or graphic	(9)
176	CTRL-1 or graphic (9) or CTRL-SHIFT-0 (9)	
177	CTRL-4 or graphic	(9)
178	CTRL-5 or graphic	(9)
179	CTRL-6 or graphic	(9)
180	CTRL-7 or graphic	(9)
181	CTRL-8 or graphic	(9)
182	CTRL-9 or graphic	(9)
183	CTRL-? or graphic	(9)
184	CTRL-CE or graphic	
185	CTRL-* or graphic	(9)
186	CTRL-] or graphic	(8)
187	CTRL-8 or graphic	
188	CTRL-- or graphic	
189	CTRL-. or graphic	(9)
190	CTRL-7 or graphic	
191	CTRL-9 or graphic	(8)
192	graphic (6)	
193	A or graphic	
194	B or graphic	
195	C or graphic	
196	D or graphic	
197	E or graphic	
198	F or graphic	
199	G or graphic	
200	H or graphic	
201	I or graphic	
202	J or graphic	
203	K or graphic	
204	L or graphic	
205	M or graphic	
206	N or graphic	
207	O or graphic	
208	P or graphic	
209	Q or graphic	
210	R or graphic	
211	S or graphic	
212	T or graphic	
213	U or graphic	
214	V or graphic	
215	W or graphic	
216	X or graphic	
217	Y or graphic	
218	Z or graphic	
219	CTRL-3 or graphic or CTRL-SHIFT-00	(9)(6)
220	CTRL-; or graphic	(6)
221	graphic	
222	pi or SHIFT-pi	
223	CTRL-back arrow or CTRL-pound or graphic(6)	
224	SHIFT-space	
225	graphic	
226	graphic	

227	graphic	
228	graphic	
229	graphic	
230	graphic	(6)
231	graphic	(6)
232	graphic	
233	graphic	(6)
234	graphic	(6)
235	graphic	
236	graphic	
237	graphic	
238	graphic	
239	graphic	
240	graphic	
241	graphic	
242	graphic	
243	graphic	
244	graphic	
245	graphic	
246	graphic	(6)
247	graphic	
248	graphic	
249	graphic	
250	graphic	(6)
251	graphic	
252	graphic	
253	graphic	
254	graphic	
255	RVS-pi	

Notes:-

1) No key gives null not 0.
   CTRL, SHIFT, Undefined F-keys and STOP are not detectable in the same way as other keys.

2) Only visible in quotes mode.

3) When next in direct mode this will force:

   DLOAD "*" and RUN

   into the keyboard buffer.

4) Shift Carriage Return in any mode.

5) Insert in any mode

6) It is possible to generate two graphic characters here.

7) It is possible to generate three graphic characters here

8) It is possible to generate four graphic characters here.

9) The numeric keypad key, not the main keyboard key.

APPENDIX H (Mainly for use with screen display)

POKE/PEEK Code	Character/function
0	@
1	a or A
2	b or B
3	c or C
4	d or D
5	e or E
6	f or F
7	g or G
8	h or H
9	i or I
10	j or J
11	k or K
12	l or L
13	m or M
14	n or N
15	o or O
16	p or P
17	q or Q
18	r or R
19	s or S
20	t or T
21	u or U
22	v or V
23	w or W
24	x or X
25	y or Y
26	z or Z
27	[
28	pound
29	]
30	↑
31	back arrow
32	space
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7

56	8	
57	9	
58	:	
59	;	
60	<	
61	=	
62	>	
63	?	
64	graphic	
65	A or graphic	
66	B or graphic	
67	C or graphic	
68	D or graphic	
69	E or graphic	
70	F or graphic	
71	G or graphic	
72	H or graphic	
73	I or graphic	
74	J or graphic	
75	K or graphic	
76	L or graphic	
77	M or graphic	
78	N or graphic	
79	O or graphic	
80	P or graphic	
81	Q or graphic	
82	R or graphic	
83	S or graphic	
84	T or graphic	
85	U or graphic	
86	V or graphic	
87	W or graphic	
88	X or graphic	
89	Y or graphic	
90	Z or graphic	
91	graphic	
92	graphic	(1)
93	graphic	
94	pi	
95	graphic	(1)
96	SHIFT-space	
97	graphic	
98	graphic	
99	graphic	
100	graphic	
101	graphic	
102	graphic	(1)
103	graphic	
104	graphic	(1)
105	graphic	(1)
106	graphic	
107	graphic	
108	graphic	
109	graphic	
110	graphic	
111	graphic	
112	graphic	

113	graphic	
114	graphic	
115	graphic	
116	graphic	
117	graphic	
118	graphic	
119	graphic	
120	graphic	
121	graphic	
122	graphic	(1)
123	graphic	
124	graphic	
125	graphic	
126	graphic	
127	graphic	
128	RVS-@	
129	RVS-A/a	
130	RVS-B/b	
131	RVS-C/c	
132	RVS-D/d	
133	RVS-E/e	
134	RVS-F/f	
135	RVS-G/g	
136	RVS-H/h	
137	RVS-I/i	
138	RVS-J/j	
139	RVS-K/k	
140	RVS-L/l	
142	RVS-N/n	
143	RVS-O/o	
144	RVS-P/p	
145	RVS-Q/q	
146	RVS-R/r	
147	RVS-S/s	
148	RVS-T/t	
149	RVS-U/u	
150	RVS-V/v	
151	RVS-W/w	
152	RVS-X/x	
153	RVS-Y/y	
154	RVS-Z/z	
155	RVS-[	
156	RVS-pound	
157	RVS-]	
158	RVS-↑	
159	RVS-back arrow	
160	RVS-SHIFT space	
161	RVS-!	
162	RVS-"	
163	RVS-#	
164	RVS-$	
165	RVS-%	
166	RVS-&	
167	RVS-'	
168	RVS-(	
169	RVS-)	
170	RVS-*	

171	RVS-+	
172	RVS-,	
173	RVS--	
174	RVS-.	
175	RVS-/	
176	RVS-0	
177	RVS-1	
178	RVS-2	
179	RVS-3	
180	RVS-4	
181	RVS-5	
182	RVS-6	
183	RVS-7	
184	RVS-8	
185	RVS-9	
186	RVS-:	
187	RVS-;	
188	RVS-<	
189	RVS-=	
190	RVS->	
191	RVS-?	
192	RVS-graphic	
193	RVS-graphic	
194	RVS-graphic	
195	RVS-graphic	
196	RVS-graphic	
197	RVS-graphic	
198	RVS-graphic	
199	RVS-graphic	
200	RVS-graphic	
201	RVS-graphic	
202	RVS-graphic	
203	RVS-graphic	
204	RVS-graphic	
205	RVS-graphic	
206	RVS-graphic	
207	RVS-graphic	
208	RVS-graphic	
209	RVS-graphic	
210	RVS-graphic	
211	RVS-graphic	
212	RVS-graphic	
213	RVS-graphic	
214	RVS-graphic	
215	RVS-graphic	
216	RVS-graphic	
217	RVS-graphic	
218	RVS-graphic	
219	RVS-graphic	
220	RVS-graphic	
221	RVS-graphic	
222	RVS-pi	
223	RVS-graphic	(1)
224	RVS-SHIFT space	
225	RVS-graphic	
226	RVS-graphic	
227	RVS-graphic	

228	RVS-graphic	
229	RVS-graphic	
230	RVS-graphic	(1)
231	RVS-graphic	
232	RVS-graphic	
233	RVS-graphic	(1)
234	RVS-graphic	
235	RVS-graphic	
236	RVS-graphic	
237	RVS-graphic	
238	RVS-graphic	
239	RVS-graphic	
240	RVS-graphic	
241	RVS-graphic	
242	RVS-graphic	
243	RVS-graphic	
244	RVS-graphic	
245	RVS-graphic	
246	RVS-graphic	
247	RVS-graphic	
248	RVS-graphic	
249	RVS-graphic	
250	RVS-graphic	(1)
251	RVS-graphic	
252	RVS-graphic	
253	RVS-graphic	
254	RVS-graphic	
255	RVS-graphic	

Note:-

(1) Two or more graphic characters are possible with this code.

BIBLIOGRAPHY

1) "Compute's first book of PET/CBM". Published by Compute. General introduction to CBM business computers.

2) "BASIC Basic-English dictionary". By Larry Noonan. Published by Dilithium Press (USA).

3) "MOS Programming Manual". MOS Technology. Faulk Baker Associates.

4) "Library of PET subroutines". By N. Hampshire. Published by Hayden Books. Many of the routines would need adapting for the 700, but the ideas are sound.

5) "PET graphics". By N. Hampshire. Published by Hayden Books. Many of the routines would need adapting for the 700, but the ideas are sound.

6) "CBM Personal Computer Guide". By C. Donahue. Published by Osborne/McGraw Hill. Good grounding in Commodore BASIC 4.0. Does not cover the 700 extensions.

7) "CBM Professional Computer Guide". By A. Osborne, J. Strasma and E. Strasma. Published by Osborne/McGraw Hill (USA). Similar to 6) above, but more business orientated.

8) "PET and IEEE 488 bus (GPIB)". By E. Fisher and C. Jensen. Published by Reston (USA). The IEEE interface as used by CBM machines. Examples would need adapting to run on a 700.

9) "PET and the IEEE". By A. Osborne and C. Donahue. Published by Osborne/McGraw Hill (USA). Comment as for 8) above.

10) "Programming the 6502". By R. Zaks. Published by Sybex (USA). Good introduction to the 6502 which is very similar to the 6509 in the 700. Use in conjunction with the 700 Kernal Manual.

11) "Programming the PET/CBM". By R. West. Published by Level Limited (UK). Excellent book, but need some adaption for the 700 - especially the machine code section. Does not cover the extensions to BASIC 4.0 in the 700 BASIC 4.0+.

12) "Commodore 64 Programmer's Reference Guide". Commodore. Included here for the users of 500 machines, and for the SID chip information.

13) "Applications Catalogue". Commodore (UK). A list of business packages for all CBM machines including the 700.

14) "Microprocessor Interfacing Techniques". - Third Edition. By R. Zaks and A. Lesea. Published by Sybex (USA). Interfacing techniques in general with examples.

15) "6502 Assembly Language Programming". By L. Leventhal. Published by McGraw Hill (USA). See comments 10) above.

16) "The Art of Computer Programming, Volume 1". By D. Knuth. Published by Addison-Wesley (USA). This volume is about Fundamental Algorithms. (Second editiion.)

17) "The Art of Computer Programming, Volume 3". By D. Knuth. Published by Addison-Wesley (USA). This volume is about Sorting and Searching.

Note: This is simply a list of books. The reader must decide whether they are useful or not. Commodore (UK) does not endorse or subsidise any of the titles in this list, neither does the author of this manual recommend any of the titles.

NOTICE

The Information contained in this document provides a specification for the Kernal in 6509 based machines. No responsibility is assumed by Commodore for ommissions or errors. The information contained in this guide is subject to change without notice.

TABLE OF CONTENTS

INTRODUCTION

The following list of Kernal routines is intended to facilitate the movement of assembly language programs between CBM machines. Programs written in Commodore BASIC have generally been upward compatible. It is our desire to present a list of assembly language I/O routines that the programmer can use for utilities, interpreters, assemblers and compilers. By using only routines in this list, resulting programs can be I/O independent, and hopefully independent of hardware of future machines. To create new software versions at that time, only a new assembly, with perhaps a different origin, would be required.

Please note that no routines are supported for data structures or mathematics. Both these features are subject to great changes. The user program should handle its own data and communicate with the I/O routines through the standard channels.

1. POWER-UP ACTIVITIES

Upon reset the Kernal initialises the stack pointer to $FF, clears the decimal mode flag, and checks locations $0F03FA and $0F03FB for warm start information. if location $0F03FA=$xx and location $0F03FB=$xx then the initialisation phase is skipped and a JMP ($0F03F8) occurs. If these locations differ, the screen editor is initialised followed by a check for USER ROMS. BASIC is expected in most machines starting at location $0F8000 (the BASIC sequence at $0F8006 is $C3,$C2,$CD,$38).

C B M 8

The standard Kernal sequence is to initialise I/O, clear system RAM, test USER RAM, initialise Kernal variables, inititalise Screen Editor, then set the WARM reset variables.

The I/O initialisation will reset all the standard system devices to a non-active state, set the keyboard lines to a stop-key check state, set the TOD for the proper line frequency and send IFC (reset) to devices connected to the IEEE bus.

The system RAM $0F0002-$0F0101 and $0F0200-$0F03F7 is set to zero. The USER RAM is tested starting at segment/bank 0 on the 500 and segment/bank 1 on the 700. An $55 and $AA pattern is tried in each location and then the original data is restored. If a RAM failure occurs with in a segment, the Top-of-memory pointer will be set to the segment preceding the failure. The test will continue until a non-RAM segment is found, thus all 13 or 14 possible RAM segments can be tested. The RS232 input buffer is also flagged as unassigned by this routine, this being allocated by the OPEN file system.

Locations $0F0090-$0F00FE are used by the Kernal for its variables and page zero indirects. In addition, absolute locations from $0F0XXX to $0F0XXX are used for other variable storage.

2. USER CALLABLE KERNAL ROUTINES

Name	Adr	Function	Section
ACPTR	$FFA5	Input byte from IEEE bus	1
CHKIN	$FFC6	Open channel for input	2
CHKOUT	$FFC9	Open channel for output	3
CHRIN	$FFCF	Input character from channel	4
CHROUT	$FF02	Output character to channel	5
CIOUT	$FFA8	output byte to IEEE bus	6
CLALL	$FFE7	Close all files	7
CLOSE	$FFC3	Close logical file	8
CLRCHN	$FFCC	Close input and output channel	9
GETIN	$FFE8	Get character from keyboard queue	10
IOBASE	$FFF3	Return base address of I/O	11
LISTEN	$FFB1	Command IEEE device to listen	12
LKUPLA	$FF8D	Lookup device data on LA	13
LKUPSA	$FF8A	Lookup device data on SA	14
LOAD	$FFD5	Load RAM from device	15
MEMBOT	$FF9C	READ/SET bottom of memory	16
MEMTOP	$FF99	READ/SET top of memory	17
OPEN	$FFC0	Open logical file	18
PLOT	$FFF0	READ/SET X,Y cursor position	19
RDTIM	$FFDE	Read real time clock	20
READST	$FFB7	Read I/O status word	21
RESTOR	$FF87	Restore old I/O vectors	22
SAVE	$FFD8	Save RAM to device	23
SCNKEY	$FF9F	Scan keyboard	24
SCREEN	$FFED	Return X,Y organisation of screen	25
SECOND	$FF93	Transmit secondary command after listen	26
SETLFS	$FFBA	Set logical, first, second addresses	27
SETMSG	$FF90	Control Kernal messages	28
SETNAM	$FFBD	Set file name information	29
SETTIM	$FFDB	Set real time clock	30
SETTMO	$FFA2	Set timeout on IEEE	31
STOP	$FFE1	Check stop key	32
TALK	$FFB4	Command IEEE device to talk	33
TKSA	$FF96	Send secondary after talk	34
UOTIM	$FFEA	Increment real time clock	35
UNLSN	$FFAE	Command IEEE bus to unlisten	36
UNTLK	$FFAB	Command IEEE bus to untalk	37
VECTOR	$FF84	Read/set vectored I/O	38

Format of Function Descriptions

The following conventions are used in describing the Kernal entry points:-

Function name: This is a symbol assigned to the memory location for reference only. It is used to develop a standard naming convention but user is free to use own mnemonic.

Call address: This is the subroutine call address of the Kernal routine. It is given in hexadecimal. If the address is followed by an (I) that means the address is indirected. (VECTORS in page 3)

Communication registers: Registers listed here are used to pass parameters to and from kernal routines.

Affected registers on return: Registers listed here are no longer valid, or changed by actions within the routines. Many calls may return no valid registers if an error occurs (carry-set return).

Preparatory routines: Sometimes data must be set up before a Kernal routine can function. Routines to set up this data are listed here.

Error returns: Where applicable, a return from the Kernal with carry set means that the accumulator contains the number of an error encountered in processing.

Stack requirements: This is the actual number of stack bytes used to hold the return address or any other bytes used on the stack by the Kernal subroutine.

Description: A short tutorial on each Kernal routine function is given here.

2.1

Function name: ACPTR

Call address: $FFA5

Communication registers: .A

Affected registers on return: .A

Preparatory routines: TALK, TKSA

Error returns: See READST

Stack requirements: 4

Description:

This routine handshakes a byte off the IEEE bus. The data is returned in the accumulator. it is assumed that the device has been told to TALK and it is possible that a secondary command has been sent by TKSA.

```
Example: JSR ACPTR
         STA DATA
```

2.2

Function name CHKIN

Call address: $FFC6 (I)

Communication registers: .X

Affected registers on return: all

Preparatory routine: OPEN

Error returns: Ø,3,5,6

Stack requirements: 6

Description:

Opening a channel for input.

Assuming that a file has been opened by subroutine OPEN, it can be opened as an input channel. Of course the characteristics of the device will be determine if it is valid to do so. This subroutine must be executed before subroutines CHRIN or GETIN are executed for a device other than the keyboard. If input form the keyboard is desired, and there is no association to the logical file number by a previous open file, then the call to this subroutine may be dispensed with.

On the IEEE this subroutine results in sending a talk address followed by a secondary address if one was specified in the open subroutine.

Example:
```
;OPEN LOGICAL FILE 2 FOR INPUT
LDX #2
JSR CHKIN
```

2.3

Function name: CHKOUT

Call address: $FFC9 (I)

Communication registers: .X

Affected registers on return: all

Preparatory routines: OPEN

Error returns: 0,3,5,7

Stack requirements: 10

Description:

Open channel for output.

Assuming that a file has been opened by subroutine OPEN, it can be opened as an output channel. Of course, the characteristics of the device will determine if it is valid to do so. This subroutine must be executed before subroutine CHROUT is executed for a device other than the CBM CRT. If output to the CRT is desired, and there is no association to an open file by logical file number, then the call to this subroutine may be dispensed with.

On the IEEE this subroutine results in sending a listen address followed by a secondary address if one was specified in the OPEN subroutine.

```
Example: ;OPEN LOGICAL FILE 3 AS OUTPUT CHANNEL
         LDX #3
         JSR CHKOUT
```

CBM Kernal - KERNAL ROUTINES

2.4

Function name: CHRIN

Call address: $FFCF (I)

Communication registers: .A

Affected registers on return: .A

Preparatory routines: None

Error returns: See READST

Stack requirements: dependant on external media

Description:

Input character from channel.

A call of this routine will return a character of data from the channel set up by a call to subroutine CHKIN or the default input channel if no other has been set up. Data is returned in the accumulator. The channel remains open after the call. In the case of the keyboard device, the cursor is turned on and continues to blink until carriage return is typed and then characters on the line are returned one by one by calls to this routine. Finally carriage return is sent and the process begins again.

```
Example:  JSR CHRIN
          STA DATA
```

CBM Kernal - KERNAL ROUTINES

2.5

Function name: CHROUT

Call address: $FFD2 (I)

Communication registers: .A

Affected registers on return: .A

Preparatory routines: None

Error returns: Ø and see READST

Stack requirements: dependant on external media

Description:

Output character to channel.

The data to be output is loaded into the accumulator. A call to CHKOUT sets up the output channel or if this call is omitted, data is sent to the default device which is number 3, CRT. The character can be transmitted to multiple devices on the IEEE if a clear channel is not performed after the corresponding open channel for output.

```
Example: ;CMD 4,"A";
         LDX #4      ;LOGICAL FILE #4
         JSR CHKOUT  ;OPEN CHANNEL OUT
         LDA #'A
         JSR CHROUT  ;SEND CHARACTER
```

CBM Kernal - KERNAL ROUTINES

2.6

Function name: CIOUT

Call address: $FFA8

Communication registers: .A

Affected registers on return: none

Preparatory routines: LISTEN, [SECOND]

Error returns: See READST

Stack requirements: 7

Description:

Handshake byte out.

The accumulator is loaded with a byte to handshake as data on the IEEE bus. A device must be listening or status reflects a timeout. One character is always buffered by this routine. When an UNLSN subroutine call is made, the buffered character is sent with EOI asserted, and then the UNLSN is sent.

CBM Kernal - KERNAL ROUTINES

2.7

Function name: CLALL

Call address: $FFE7 (I)

Communication registers: .A .SP

Affected registers on return: all

Preparatory routines: None

Error returns: None

Stack requirements: depends on external media

Description:

Carry bit clear: Close all files and reset I/O channels. The pointers into the open file table are reset. Additionally, CLRCHN is called to reset the I/O channels.

```
Example: ;USED AS START OF EXECUTION
         JSR CLRCHN ;CLOSE FILES
         JMP RUN ;BEGIN EXECUTION
```

Carry bit set : Close all files that FA (device #) is sent in .A. This will search the table and perform the CLOSE call for each file associated with the device #, but will abort on the first error return (checks the carry bit not the STATUS).

CBM Kernal - KERNAL ROUTINES

2.8

Function name: CLOSE

Call address: $FFC3 (I)

Communication registers: .A

Affected registers on return: all

Preparatory routines: None

Error returns: .A and READST

Stack requirements: depends on external media

Description:

Close a logical file.

When all I/O has completed to a file, call this subroutine with the accumulator loaded with the logical file number used in the OPEN subroutine.

```
Example: ;close logical file 15
         LDA #15
         JSR CLOSE
```

CBM Kernal - KERNAL ROUTINES

2.9

Function name: CLRCHN

Call address: $FFCC (I)

Communication registers: None

Affected registers on return: .A,.X

Preparatory routines: None

Error returns: None

Stack requirements: 9

Description:

Clear channel.

After opening a channel and performing I/O, this routine closes all open channels and restores the default channels. Input is device Ø and output is 3. This routine may be called optionally by the programmer. An untalk is sent to clear the input channel if the device is on the IEEE. An unlisten is sent to clear the output channel. By not calling this routine and leaving a listener addressed on the IEEE, multiple devices can receive data on the bus. An example would be to address the printer to listen and the disk to talk.

Example: JSR CLRCHN

CBM Kernal - KERNAL ROUTINES

2.10

Function name: GETIN

Call address: $FFE4 (I)

Communication registers: .A

Affected registers on return: .A

Preparatory routines: None

Error returns: 0 and see READST

Stack requirements: depends on circumstances when called.

Description:

Get buffered character from keyboard.

This subroutine removes one character from the keyboard queue and returns an ASCII value in the accumulator. If the queue is empty, the value returned will be zero. Characters are put into the queue by an interrupt driven scan which calls SCNKEY.

```
Example: ;WAIT FOR CHARACTER
         WAIT JSR GETIN
              CMP #0
              BEQ WAIT
```

2.11

Function name: IOBASE

Call address: $FFF3

Communication registers: .X,.Y

Affected registers on return: .X,.Y

Preparatory routines: None

Error returns: None

Stack requirements: 2

Description: Returns address of page containing I/O in X,Y. This can be used with an offset to access memory mapped I/O devices in the 700 and 500. In the 6509 Kernals all I/O is in segment $0F. This function and subsequent register accesses are machine dependent.

Example:
```
JSR IOBASE
STX POINT
STY POINT + 1
LDA #0
LDY #2
STA (POINT)Y
```

2.12

Function name: LISTEN

Call address: $FFB1

Communication registers: .A

Affected registers on return: .A

Preparatory routines: None

Error returns: See READST

Stack requirements: 10

Description:

Listen with attention.

The accumulator is loaded with a device number between 0 and 30. This subroutine ORs in bits to convert this device number to listen address and then transmits this data as a command on the IEEE bus.

```
Example: ;COMMAND DEVICE #8 TO LISTEN
         LDA #8
         JSR LISTEN
```

2.13

Function name: LKUPLA

Call address: $FF8D

Communication registers: .A,.X,.Y

Affected registers on return: .A,.X,.Y

Preparatory routines: None

Error returns: carry-set is no LA found

Stack requirements: 4

Description:

Match file parameters keyed on logical address. Routine is called with the LA in .A. It returns with either an error (no match = carry set) or the FA in .X and the SA in .Y. Also clears the STATUS variable.

```
Example: ;FIND DEVICE FOR LA=2
         LDA #2
         JSR LKUPLA
```

2.14

Function name: LKUPSA

Call address: $FF8A

Communication registers: .A,.X,.Y

Affected registers on return: .A,.X,.Y

Preparatory routines: None

Error returns: carry-set is no SA found

Stack requirements: 4

Description:

Match file parameters keyed on secondary address. Routine is called with SA in .Y. Returns either with error (no match = carry set) or LA in .A and FA in .X.

```
Example: ;FIND DEVICE FOR SA=2
         LDY #2
         JSR LKUPSA
```

2.15

Function name: LOAD

Call address: $FFD5 (I)

Communication registers: .A,.X,.Y

Affected registers on return: all

Preparatory routines: SETLFS, SETNAM

Error returns: 0,4,5,8,9, see READST

Stack requirements: depends on external media

Description:

Load from device into RAM. On call, .A(bit 7)=0 for load, .A(bit 7)=1 for verify, .A(bits 0123)=start segment. Registers .X=start address low and .Y=start address high, are used to determine the load address. If .X and .Y are equal to $FF, then the load begins where the header has specified. On return (.A,.X,.Y) is highest RAM address loaded.

```
Example: LDX DEVICE
         LDA FILENO
         LDY CMD
         JSR SETLFS
         LDA #$0F         ;this code is in segment F
         STA ZNAME+2      ;zname is an z-page 3 byte pointer
         LDA #>NAME
         STA ZNAME+1
         LDA #<NAME
         STA ZNAME
         LDA #NAME1-NAME
         LDX #ZNAME       ;z-page location of 3 byte pointer
         JSR SETNAM
         LDA #%00000000   ;flag load, start in seg 0, for a 500
                          ;for a 700 use %00000001 for Basic bank
         LDX #$FF         ;default load, to header address
         LDY #$FF
         JSR LOAD
         STX VARTAB       ;end of load
         STY VARTAB+1
         JMP START
 NAME    .BYT 'FILE NAME'
 NAME 1  ;
```

2.16

Function name: MEMBOT

Call address: $FF9C

Communication registers: .A,.X,.Y,.SP

Affected registers on return: none on write, all on read

Preparatory routines: None

Error returns: None

Stack requirements: 2

Description: A call of this subroutine with carry bit set causes a read of the pointer to the lowest byte of RAM and this address is returned in .A, .X and .Y. The initial value is determined by system configuration.

Calling this routine with carry clear causes a transfer of the bytes in .X and .Y to the low and high bytes of this pointer, with .A containing the segment number.

Example:
```
;MOVE BOTTOM OF MEMORY UP 1 PAGE
SEC
JSR MEMBOT; Get
INY
CLC
JSR MEMBOT; Put
```

2.17

Function name: MEMTOP

Call address: $FF99

Communication registers: .A,.X,.Y,.SP

Affected registers on return: none on write, all on read

Preparatory routines: None

Error returns: None

Stack requirements: 2

Description: When this routine is called with carry set, the pointer to the top of RAM is read into .A, .X and .Y.

A call with carry clear will copy the contents of .A, .X and .Y into this pointer. The space between the MEMTOP pointer and the absolute top of avaliable RAM is the space where KERNAL buffers are allocated. If one wishes to protect user software by this pointer allowances for buffer demands should be made.

2.18

Function name: OPEN

Call address: $FFC0 (I)

Communication registers: .SP

Affected registers on return: all

Preparatory routines: SETLFS, SETNAM

Error returns: 0,1,2,4,5,6

Stack requirements: depends on external media

Description:

Open logical file. Arguments are set up by the external routine calls SETLFS and SETNAM which should be called before this routine.

A carry-set call opens a temporary channel on the IEEE system, with no file table manipulation, which is used to send disk commands via the filename area to our IEEE disk units.

The carry-clear entry will perform normal open operations and leave table information for other I/O calls (CHKIN, CHKOUT, CHRIN, CHROUT, CLOSE).

See overleaf for example.

Example: This is an implementation of the BASIC statement:

OPEN 15,8,15,"I/O"

```
        LDA #$ØF          ;set pointer to name in zero page
        STA ZNAME+2       ;the name is in the ROM segment
        LDA #>NAME
        STA ZNAME+1
        LDA #<NAME
        STA ZNAME
        LDA #NAME2-NAME; ;LENGTH
        LDX #ZNAME
        JSR SETNAM
        LDA #15
        LDX #8
        LDY #15
        JSR SETLFS
        CLC
        JSR OPEN
        NAME.BYT 'I/O'
        NAME2
```

2.19

Function name: PLOT

Call address: $FFFØ

Communication registers: .X,.Y,.P

Affected registers on return: all

Preparatory routines: None

Error returns: None

Stack requirements: 2

Description: A call with carry set reads the current X,Y position of the cursor on the screen into .X, .Y.

A call with carry clear moves the cursor to X,Y as determined by .X, .Y.

Example:
```
; MOVE TO 5,5
LDX #5
LDY #5
CLC
JSR PLOT
```

2.2Ø

Function name: RDTIM

Call address: $FFDE

Communication registers: .A,.X,.Y

Affected registers on return: all

Preparatory routines: None

Error returns: None

Stack requirements: 2

Description:

Read time. The system clock can be read at any time. The system clock in the 5ØØ and 7ØØ is based upon line frequency. The values returned by this call are as follows:

```
Registers: .A  bit 7 = AM/PM indicator
               bit 6 = bit 3 bcd tenths of a second
               bit 5 = bit 2 bcd tenths of a second
               bit 4 to bit Ø = bcd hours


           .X  bit 7 = bit 1 bcd tenths of a second
               bit 6 to bit Ø = bcd minutes


           .Y  bit 7 = bit Ø bcd tenths of a second
               bit 6 to bit Ø = bcd seconds
```

2.21

Function name: READST

Call address: $FFB7

Communication registers: .A

Affected registers on return: .A

Preparatory routines: None

Error returns: None

Stack requirements: 2

Description:

Returns current I/O status. Usually checked after initiating any new communication to a channel. Each of the bits in the byte returned contain data. See the table overleaf.

```
Example: ;CHECK FOR DEVICE NOT PRESENT ON IEEE
         JSR READST
         AND #128  ;check DNP bit 7
         BNE DNP   ;branch if device not present
```

ST Bit Position	ST Numeric Value	Cassette* Read	IEEE/RW	Tape Verify* + Load
Ø write	1		Time out	
1 read	2		Time out	
2	4	Short block		Short block
3	8	Long block		Long block
4	16	Unrecoverable read error		Any mismatch
5	32	Checksum error		Checksum error
6	64	End of file	EOI line	
7	-128	End of tape	Device not present	End of tape

* 5ØØ only. THE 700 HAS NO CASSETTE I/O.

2.22

Function name: RESTOR

Call address: $FF87

Communication registers: None

Affected registers on return: all

Preparatory routines: None

Error returns: None

Stack requirements: 2

Description:

Restore default vector values for system subroutines and interrupts. See VECTOR for reading and altering contents.

Example: JSR RESTOR

2.23

Function name: SAVE

Call address: $FFD8

Communication registers: .X,.Y

Affected registers on return: all

Preparatory routines: SETLFS, SETNAM

Error returns: 0,5,8,9 and see READST

Stack requirements: dependant on external media

Description: Saves memory form zero page pointer set by .X to zero page pointer set by .Y. A file name is not required for device 1 (500 cassette machines) but an error condition exists for any other device save without a file name. Device 0 (keyboard), device 2 (RS232), and device 3 (screen) are not defined for SAVE.

Example:

```
        LDA #1          ;DEVICE=1:CASSETTE on a 500:
                        ;Illegal on a 700!
        JSR SETLFS
        LDA #0          ;NO FILE NAME
        JSR SETNAM
        LDX    #STARTV  ;START VECTOR (3 BYTES (LOW)(HIGH)(SEG#))
        LDY #ENDV       ;END VECTOR (3 BYTES)
        JSR SAVE
```

2.24

Function name: SCNKEY

Call address: $FF9F

Communication registers: None

Affected registers on return: all

Preparatory routines: None

Error returns: None

Stack requirements: 5

Description: Scan the keyboard. This is the same subroutine as is called by the interrupt handler. If a key is down, its value, if any is placed in the keyboard queue.

```
Example: GET JSR SCNKEY  ;SCAN KEYBOARD
             JSR GETIN   ;GET CHARACTER
             CMP #Ø      ;IS IT NULL?
             BEQ GET     ;YES...SCAN AGAIN
             JSR CHROUT  ;PRINT IT
```

2.25

Function name: SCREEN

Call address: $FFED

Communication registers: .X,.Y

Affected registers on return: .X,.Y

Preparatory routines: None

Error returns: None

Stack requirements: 2

Description: Returns constant organization of screen e.g. 40 columns in .X and 25 lines in .Y, or 80 in .X and 25 in .Y.

Example:
```
JSR SCREEN
STX MAXCOL
STY MAXROW
```

2.26

Function name: SECOND

Call address: $FF93

Communication registers: .A

Affected registers on return: .A

Preparatory routines: LISTEN

Error returns: See READST

Stack requirements: 8

Description:

Secondary address after LISTEN. This routine cannot be used to send a secondary address after a TALK.

Example:
```
;DEVICE #8 WITH COMMAND #15
LDA #8
JSR LISTEN
LDA #15
JSR SECOND
```

2.27

Function name: SETLFS

Call address: $FFBA

Communication registers: .A,.X,.Y

Affected registers on return: all

Preparatory routines: None

Error returns: None

Stack requirements: 2

Description:

Setting logical file number, device address, and command.

The logical file number is used as a key by the system to access data stored in a table by the open file subroutine. The device address ranges from 0 to 30 and corresponds to the devices on the table overleaf.

Load the accumulator with the logical file number, X index with the device number, and Y index with the command. The command is sent as a secondary address on the IEEE following the device number during an attenttion sequence. If the programmer desires no secondary address to be sent, load Y index with a 255.

Example: For logical file 32, device #4, and no command:

```
LDA #32
LDX #4
LDY #255
JSR SETLFS
```

```
 0  Keyboard
 1  Cassette #1 (500 only - illegal on a 700)
 2  RS232
 3  CRT display
 4  IEEE printer
 5  IEEE Modem or Second printer
 6  IEEE plotter
 8  CBM IEEE disk-drive
 9  CBM IEEE Second or Hard disk drive.
10  and above are user devices
```

Device numbers 4 or greater correspond to devices on the IEEE bus.

2.28

Function name: SETMSG

Call address: $FF9Ø

Communication registers: .A

Affected registers on return: None

Preparatory routines: None

Error returns: None

Stack requirements: 2

Description:

This routine controls the printing of error and diagnostic messages by the kernal. It is called by placing a value in the accumulator. Bits 6 and 7 of this value control the message printing. Bit 7 controls the printing of error messages from the kernal. If it is set then messages like "I/O ERROR #4" will appear. Bit 6 controls the printing of control messages.

```
Example: LDA #$4Ø
         JSR SETMSG  ;turn on diagnostics
         ...
         LDA #Ø
         JSR SETMSG  ;turn off all kernal messages
```

2.29

Function name: SETNAM

Call address: $FFBD

Communication registers: .A,.X

Affected registers on return: .A

Preparatory routines: None

Stack requirements: None

Description:

If a file will be opened without a file name, the file name length must be set to zero. Load the accumulator with the length, X index with a zero page pointer value ((low)(high)(seg #)), which points to the filename in memory. The file name address can be any valid memory address where the string of characters corresponding to the file name are stored.

```
Example: LDA #NAME2-NAME  ;load length of file name
         LDX #<NAME       ;load address of disk file name
         LDY #>NAME
         JSR SETNAM
```

2.30

Function name: SETTIM

Call address: $FFDB

Communication registers: .A,.X,.Y

Affected registers on return: all

Preparatory routines: None

Error returns: None

Stack requirements: 4

Description:

Set time-of-day.

```
Registers: .A  bit 7 = AM/PM indicator
               bit 6 = bit 3 bcd tenths of a second
               bit 5 = bit 2 bcd tenths of a second
               bit 4 to bit 0 = bcd hours

           .X  bit 7 = bit 1 bcd tenths of a second
               bit 6 to bit 0 = bcd minutes

           .Y  bit 7 = bit 0 bcd tenths of a second
               bit 6 to bit 0 = bcd seconds
```

2.31

Function name: SETTMO

Call address: $FFA2

Communication registers: .A

Affected registers on return: None

Preparatory routines: None

Error returns: None

Stack requirements: 2

Description:

Set timeout flag.

When the accumulator contains a 0 in bit 7, timeouts are enabled by this routine. A 1 in bit 7 disables timeouts. Timeouts are a way that the CBM can poll an IEEE device for data without hanging in a handshake sequence. The device must respond to DAV within 64 milliseconds. The CBM disks use the timeout feature to communicate a file not found status in OPEN.

```
Example: ;DISABLE TIMEOUT
         LDA #0
         JSR SYS21
```

2.32

Function name: STOP

Call address: $FFE1 (I)

Communication registers: None

Affected registers on return: .A,.X

Preparatory routines: UDTIM

Error returns: None

Stack requirements: 2

Description:

Check for stop key. If stop key is down, clear all channels to default.

This routine clears all I/O channels to default values (CLRCHN call) and returns with the Z flag set, if the STOP key on the keyboard was pressed when the UDTIM routine was called. All other flags are maintained. If the stop key is not pressed then the accumulator contains a byte corresponding to the last row of the keyboard scan. The user can check for certain other keys in this manner.

```
Example: JSR STOP
         BNE *+5     ;NOT DOWN
         JMP READY   ;=...STOP
```

2.33

Function name: TALK

Call address: $FFB4

Communication registers: .A

Affected registers on return: .A

Preparatory routines: None

Error returns: See READST

Stack requirements: 7

Description:

Talk with attention.

The accumulator is loaded with a device number between Ø and 3Ø. This subroutine ORs in bits to convert this device number to a talk address and then transmits this data as a command on the IEEE bus.

```
Example: ;COMMAND DEVICE #4 TO TALK
         LDA #4
         JSR TALK
```

2.34

Function name: TKSA

Call address: $FF96

Communication registers: .A

Affected registers on return: .A

Preparatory routines: TALK

Error returns: see READST

Stack requirements: 6

Description:

Secondary address for talk.

By loading the accumulator with a value, the user sends a secondary address command over the IEEE with this subroutine. This routine can only be called after TALK. It will not work after LISTEN.

Typical values sent for secondary address:

```
  LOAD -.$61    Opens a channel #1 to access a file on the disk.
  OPEN - $6X    X ranges from Ø-15 for disk access.
```

Others values can be sent, but the range is Ø-31 for standard IEEE.

```
Example: ;DEVICE #4 TO TALK AND COMMAND #5
         LDA #4
         JSR TALK
         LDA #5
         JSR TALKSA
```

2.35

Function name: UDTIM

Call address: $FFEA

Communication registers: None

Affected registers on return: .A,.X

Preparatory routines: None

Error returns: None

Stack requirements: 2

Description: This subroutine is normally called by the keyboard interrupt routine and is used to maintain the keyboard value for the STOP key routine.

Example:
```
         JSR UDTIM   ;check latest keyboard state
         JSR STOP    ;check stop key state
         BNE *+5     ;not down
         JMP EXITS   ;stop key exit
```

2.36

Function name: UNLSN

Call address: $FFAE

Communication registers: None

Affected registers on return: .A

Preparatory routines: None

Error returns: See READST

Stack requirements: 6

Description: Unlisten IEEE device. Use of this subroutine results in an unlisten command being transmitted on the IEEE bus.

Example: JSR UNLSN

2.37

Function name: UNTLK

Call address: $FFAB

Communicatin registers: None

Affected registers on return: .A

Preparatory routines: None

Error returns: See READST

Stack requirements: 6

Description: Untalk an IEEE device. Use of this subroutine results in an untalk command being transmitted on the IEEE bus.

Example: JSR UNTALK

2.38

Function name: VECTOR

Call address: $FF84

Communication registers: .A,.X,.Y,.SP

Affected registers on return: all

Preparatory routines: None

Error returns: None

Stack requirements: 2

Description:

A call of this routine with the carry bit set will read the current contents of the RAM vectors and put them in a list pointed at by (.A,.X,.Y).

When this routine is called with carry clear, the user list pointed at by (.A,.X,.Y) is transferred to the system RAM vectors. This process requires caution in its use. The best practice is to first read the entire vector contents into the user area, alter the desired vectors, and then copy the contents back to the system.

```
Example: ;CHANGE THE INPUT ROUTINES TO NEW SYSTEM
         LDA #USERSG
         LDX #<USER
         LDY #>USER
         SEC
         JSR VECTOR   ;read old vectors
         LDA #<MYINP  ;change input
         STA USER+1Ø
         LDA #>MYINP
         STA USER+11
         LDA #USERSG
         LDX #<USER
         LDY #>USER
         CLC
         JSR VECTOR   ;alter system
         ...
    USER *=*+26
```

3. KERNAL MONITOR FUNCTIONS

```
:  address [BY] [BY] [BY] [BY] [BY] [BY] [BY] [BY]

;  PC.  IRQ [SR AC XR YR SP]

R

M  address [address]

G  [address]

L  ["name"[,device]]

S  "name", device, long-address, long-address

Z

U  [device]

V  segment#

@  [disk command]

name
```

```
address           - hex value range $0000-$FFFF
long-address      - hex value range $000000-$0FFFFF
name              - ascii string in quotes less than
                    16 characters long.
device            - hex value range $00-$1F
segment#          - hex value range $00-$0F
disk command      - any valid command for CBM series disk
name              - any valid CBM disk filename
PC. and IRQ       - Same as address
BY, SR, AC        - hex value range $00-$FF
XR, YR, SP        - hex value range $00-$FF
```

: -- Alter memory

This command is automatically printed onto the CRT display preceding the address and data after execution of the display memory (M) command. To alter memory in this mode, the screen editor is used to change the display to the desired bytes and the <RETURN> key is pressed. The bytes are then entered into memory starting at the address specified.

; -- Alter registers

The list of data following this command is what is actually loaded into the microprocessor hardware registers when a G command is given. This command is automatically printed on the screen preceding the current list of data when an R command is executed. The list can be edited and re-entered in the same manner as the alter memory command. See the R command for contents of the list.

R -- Display registers

This command displays the contents of a list which is loaded into the 6509 hardware registers when execution is transferred from the monitor. This command also resets the view segment register. A sample display follows:

```
 R <RETURN>
 PC   IRQ   SR  AC  XR  YR  SP
;0400 E262  01  00  FF  FF  FE
```

The abbreviations correspond to the following definitions:

```
PC  = program counter
IRQ = interrupt vector
SR  = status register
AC  = accumulator
XR  = X-index register
YR  = Y-index register
SP  = stack pointer
```

M -- Display memory within a segment

If one address is specified, 16 bytes are read ~~and~~ displayed on the screen, starting at that address. For more than one address, a range of bytes is displayed, but always the next even multiple of 16 bytes from the first. The STOP key functions to stop the list.

```
M 0400 <RETURN>
: 0400 00 00 00 AA AA AA AA AA FF FF FF FF 00 00 00 00
```

G -- GO: Commence execution

With no address specified the monitor dispatches to the location contained in the PC of the register display. If an address is given execution will dispatch to that address. If a BRK (00) has been inserted in the user code, execution will return to the monitor and a register display given with the message "BREAK". On dispatch, the registers are loaded with the contents of the register display.

L -- Load memory

No file name defaults to load from cassette #1. Device number can be 1 for cassette and 4 or greater for CBM disks. The view segment register provides the segment, while the load address is contained within the load file. Load skips locations $0X0000 and $0X0001, unless these are the starting address of the load file. The STOP key will break a program LOAD. L resets the segment register to the ROM segment 15.

S -- Save memory

A file name must be specified in quotation marks as well as device number and a starting address and an ending save address. The long address form is used. If a save is started at locations $0X0000 or $0X0001, the execution and/or indirect registers will be written out, but elsewhere in the save routine these two locatons will be ignored. S resets the segment register to the ROM segment 15.

Z -- Transfer Control to Co-processor

This will 'crash' any machine without a co-processor.

U -- Set default disk drive Number

This is for use by '@' and 'name' commands.

V -- View segment

This command sets the segment register. This register is used by the Memory display and Memory write command to specify the segment range being viewed. It is also used by the Load command to specify the start segment of a file load. The Register, Save, and Load commands reset this register to the ROM segment 15.

@ -- Disk command

The command immediately followed by <RETURN> will query the disk status buffer and print its contents on the screen.

```
@ <RETURN>
00,OK,00,00
```

If a string follows the @ then that string is transmitted as a command.

```
@ INITIALIZE 0
See the 'U' command also.
```

name -- Load and execute file

When a command cannot be matched to the list of known commands, an attempt is made to load from device #8. If the load is successful, the monitor jumps to the load start address. This command is only allowed for segment 15.

APPENDIX A

PROGRAM TO DEMONSTRATE

THE USE OF KERNAL FUNCTIONS

```
;****************************************
;* example program using kernal function*
;*                                      *
;* this program reads the directory     *
;* from a commodore disk  and prints it *
;* on the crt.  a parameter list is read*
;* and passed to the disk.  the keyboard*
;* is scanned during the list to stop   *
;* and resume the list.                 *
;****************************************
;
*          = $400
;
;on entry, last character from keyboard
;is passed in .a.
;
dir        ldx #1
           ldy #'$     ;directory command
           sty $200    ;built string in buffer
           bne dir15   ;branch always
;
dir10      jsr $ffcf   ;input a character
dir15      cmp #$20
           beq dir10   ;span blanks
           cmp #$d
           beq dir20   ;stop on or
           sta $200,x
           inx
           bne dir10
;
;open directory as file
;
dir20      jsr $ffd2   ;echo or
           txa
           ldx #<$200
           ldy #>$200
           jsr $ffbd   ;set file name
           ldx #8      ;device#
           ldy #$60    ;floppy load command
           lda #1      ;logical file number
           jsr $ffba   ;set la,fa,sa
           jsr $ffc0   ;open file
;
;skip over junk, set line #
;
           ldy #3      ;do 3 times for start
;
wg220      ldx #1      ;logical file #
           jsr $ffc6   ;open for input
wg225      sty $d1
           jsr $ffcf   ;input a character
           sta $fd     ;save it
           jsr $ffb7   ;check status
           bne wg230   ;bad--stop
```

```
;
          jsr $ffcf  ;input a character
          sta $fe    ;save it
          jsr $ffb7  ;check status
          bne wg230  ;bad--stop
;
          ldy $d1    ;more to do?
          dey
          bne wg225  ;yes...
;
;print line number
;
          jsr decout
;
;print space
;
          lda #$20
          jsr $ffd2
;
;print rest of line
;
wg250     jsr $ffcf  ;get a character
          pha
          jsr $ffb7  ;check status
          bne wg230  ;bad...
          pla
          beq wg240  ;end of line
          jsr $ffd2  ;print it
          jmp wg250
;
;finish line
;
wg240     lda #$d
          jsr $ffd2  ;print cr
          jsr $ffcc  ;close channel
;
;check for stop key and pause
;
          jsr $ffe1  ;scan stop key
          beq wg230  ;stop...
;
          jsr $ffe4  ;scan keyboard
          beq wg260  ;nothing...
;
          cmp #$20   ;space bar?
          bne wg260  ;no...
;
wg255     jsr $ffe4  ;scan keyboard
          beq wg255  ;halt till key down
;
;do next line
;
wg260     ldy #2
          bne wg220
;
;close channel and file
```

```
wg230     lda #$d
          jsr $ffd2
          jsr $ffcc  ;close channel
          lda #1
          jsr $ffba  ;set la
          jsr $ffc3  ;close file
;
          jmp $f03e  ;go back to monitor
;
;
decout    ldx #0
          sec
dec100    lda $fd
          sbc #100
          sta #fd
          lda $fd+1
          sbc #0
          sta $fd+1
          bcc dec10a
          inx
          bcs dec100
dec10a    lda $fd
          adc #100
          sta $fd
          bcc dec10e
          inc $fd+1
dec10e    txa
          beq dec10b
          ora #$30
          jsr $ffd2
dec10b    sec
          ldy #0
dec10c    lda $fd
          sbc #10
          sta $fd
          bcc dec10d
          iny
          bcs dec 10c
dec10d    adc #10
          pha
          tya
          bne dec10f
          txa
          bec decla
dec10f    ora #$30
          jsr $ffd2
decla     pla
          ora #$30
          jmp $ffd2
          .end
```

APPENDIX B

MATHEMATICS ROUTINES

Decimal four-function math routines

```
;***************************************
;*  m     m   aaa   tttttt  hh    hh    *
;*  mm   mm  aa  a    tt    hh    hh    *
;*  mm m mm  aa  a    tt    hh    hh    *
;*  mm m mm  aaaaa    tt    hhhhhhhh    *
;*  mm   mm  aa  a    tt    hh    hh    *
;*  mm   mm  aa  a    tt    hh    hh    *
;*  mm   mm  aa  a    tt    hh    hh    *
;***************************************
         .ski 5
;*****listing date -- august 1, 1980*****
         .ski 5
;***************************************
;*                                     *
;*bcd math package                     *
;*                                     *
;*     the following routines are pro- *
;*vided: (+,-,*,/,:) dadd, dsub, dmult *
;*ddiv, and dcomp.  the routines are   *
;*set for fixed 22 digit precision with *
;*an exponent range +63 to -64.  the   *
;*mantissa is stored in eleven bytes   *
;*with the lsd in the lowest memory byte*
;*and least significant nybble.  the   *
;*exponent byte contains the exponent  *
;*two's complement and shifted left one *
;*bit.  the least significant bit of the*
;*exponent byte contains the sign of the*
;*mantissa.                            *
;*                                     *
;*copyright 1979 by john feagans       *
;***************************************
```

```
          .pag 'declarations'
;result register
;
resexp     *=*+1
reslsd     *=*+10
resmsd     *=*+1
;
;floating accumulator
;
facexp     *=*+1
faclsd     *=*+10
facmds     *=*+1
;
;argument register
;
argexp     *=*+1
arglsd     *=*+10
argmsd     *=*+1
;
;local variable for math routines
;
count      *=*+1
;
;user supplied routines for error
;
*          = $400
overr      brk  ;overflow error
dv0err     brk  ;divide by zero error
           .lib dadd
           .lib dmult
           .lib ddiv
           .lib dcomp
           .end
```

```
          .pag  'decimal add-sub'
;**decimal subtract fac=arg-fac***  1-02-80
;complement sign of fac mantissa
;
dsub      lda facexp
          eor #$01
          sta facexp
;
;**decimal add fac=fac-fac+arg**
;
;exchange arg and fac
;
dadd0     ldx #facmsd-facexp
;
;no exchange if arg 0
;
          lda argmsd
          bne dadd2
          lda facexp
          sta argexp
          jmp dadd
;
dadd2     ldy argexp,x
          lda facexp,x
          sta argexp,x
          sty facexp,x
          dex
          bpl dadd2
;
;check if both exponents same
;
dadd      lda facmsd
          bne dadd5
          lda argexp
          sta facexp
dadd5     lda facexp
          ora #1
          pha
          sec         ;for later subtracts
          eor argexp
          bpl dadd10
;
;compute # of times arg to be shifted right, make sure
;facexp>=argexp for case when exp signs --different--
;
          pla
          bmi dadd0   ;facexp<argexp
          sbc argexp
          jmp dadd20
;
;compute # of times arg to be shifted right, make sure
;facexp>=argexp for case when exp signs --same--
;
dadd10    pla
          sbc argexp
          bcc dadd0
```

```
          and #$fe
          bne dadd20
;
;if facexp=argexp, then make sure that
;abs(fac-mantissa)>=abs(arg-mantissa)
;
          pha
          ldx #0
;carry set here
          sed
          ldy #facmsd-faclsd
dadd12    lda faclsd,x
          sbc arglsd,x
          inx
          dey
          bpl dadd12
          cld
          pla
          bcc dadd0
;
;convert difference of exponents to shift count
;
dadd20    lsr a
          sta count
;
;shift arg mantissa right number times specified in count.
;
dadd30    dec count
          bmi dadd40
          ldy #3
dadd32    ldx #argmsd-arglsd
          clc
dadd34    ror arglsd,x
          dex
          bpl dadd34
          dey
          bpl dadd32
          bmi dadd30
;
;if both mantissa have same sign perform add : fac=fac+arg
;
dadd40    lda argexp
          eor facexp
          ror a
          ldx #0
          ldy #facmsd-faclsd
          sed
          bcs dadd50
;
dadd42    lda faclsd,x
          adc arglsd,x
          sta faclsd,x
          inx
          dey
          bpl dadd42
          bmi dnorm
```

```
;
dadd50    lda faclsd,x
          sbc arglsd,x
          sta faclsd,x
          inx
          dey
          bpl dadd50
;
;**decimal normalize fac with no lsr**
;
dnormz    clc
;
;**decimal normalize fac with potential lsr**
;
dnorm     cld
          bcs dnor20
;
;bail out if mantissa zero
;
          lda #0
          ldx #facmsd-faclsd
dnorm2    ora faclsd,x
          dex
          bpl dnorm2
          tax
          beq dzerof
;
;is msd significant yet?
;
dnor10    lda facmsd
          and #$f0
          bne dnor40  ;yes...done
;
;shift fac left one digit
;
          ldy #3
dnor12    ldx #0
          clc
   .      php
dnor14    plp
          rol faclsd,x
          php
          inx
          cpx #1+facmsd-faclsd
          bcc dnor14
          plp
          dey
          bpl dnor12
;
;decrement facexp with underflow protection
;
          lda facexp
          lsr a
          php
          sec
          sbc #1
```

```
          cmp #$3f
          bne dnor15
          plp
          jmp dzerof  ;underflow
dnor15    plp
          rol a
          sta facexp
          jmp dnor10
;
;shift fac right one digit
;
dnor20    ldy #3
dnor22    ldx #facmsd-faclsd
          clc
dnor24    ror faclsd,x
          dex
          bpl dnor24
          dey
          bpl dnor22
;
;make msd a 1 from carry
;
          lda facmsd
          ora #$10
          sta facmsd
;
;increment facexp guard overflow
;
          lda facexp
          lsr a
          php
          clc
          adc #1
          cmp #$40
          beq doverr  ;case $7f+$01->$80
          plp
          rol a
          sta facexp
;
dnor40    rts
;
doverr    jmp overr
;
;**put decimal zero in fac**
;
dzerof    lda #0
          ldx #facmsd-faclsd
dzero2    sta faclsd,x
          dex
          bpl dzero2
          sta facexp
          rts
          .end
```

```
          .pag 'decimal divide'
;**decimal divide fac=arg/fac
;    12-20-79
;division by zero error if fac zero
;
ddiv      lda facmsd
          bne *+5
          jmp dv0err
;
;done if arg is zero
;
          lda argmsd
          bne ddiv5
          jmp dzerof
;
;2's complement on divisor
;and save exponents
;
ddiv5     lda facexp
          eor #$fe
          clc
          adc #2
          pha
          lda argexp
          pha
;
          lda #0
          sta argexp
          sta facexp
          ldx #resmsd-reslsd
ddiv10    sta reslsd,x
          dex
          bpl ddiv10
          sta resexp
;
;is divisor greater than dividend
;
          ldy #facmsd-faclsd
          ldx #0
          sec
          sed
ddiv20    lda arglsd,x
          sbc faclsd,x
          sta arglsd,x
          inx
          dey
          bpl ddiv20
          lda argexp
          sbc facexp
          sta argexp
          cld
          php          ;decrement flag
          bcs ddiv80
;
;restore arg
;
```

```
ddiv30    ldy #facmsd-faclsd
          ldx #0
          clc
          sed
ddiv40    lda arglsd,x
          adc faclsd,x
          sta arglsd,x
          inx
          dey
          bpl ddiv40
          lda argexp
          adc facexp
          sta argexp
          cld
;
;is resmsd zero?
;
          lda resmsd
          and #$0f
          beq ddiv45
          plp
          pla
;
;adjust exponent
;
          bcs ddiv43
          sec
          sbc #2
          pha
          lsr a
          cmp #$3f
          bne ddiv41
          pla
          pla
          jmp dzerof
ddiv41    pla
ddiv43    sta argexp
          pla
          sta facexp
          jmp dmuldn
;
;shift arg mantissa left one digit
;
ddiv45    ldy #3
ddiv50    ldx #0
          clc
          php
ddiv52    plp
          rol arglsd,x
          php
          inx
          cpx #1+argmsd-arglsd
          bcc ddiv52
          plp
          rol argexp
```

```
;shift res mantissa left one digit
;
         ldx #0
         clc
         php
ddiv62   plp
         rol reslsd,x
         php
         inx
         cpx #1+resmsd-reslsd
         bcc ddiv62
         plp
         rol resexp
         dey
         bpl ddiv50
;
;is divisor greater than dividend
;


ddiv62   ldy #facmsd-faclsd
         ldx #0
         sec
         sed
ddiv72   lda arglsd,x
         sbc faclsd,x
         sta arglsd,x
         inx
         dey
         bpl ddiv72
         lda argexp
         sbc facexp
         sta argexp
         cld
         bcc ddiv30
;
;increment reslsd
;
ddiv80   inc reslsd
         bne ddiv70
         .end
```

```
           .pag 'decimal compare'
;decimal compare arc:fac
;december 31, 1979
;
;.a= 1, c=Ø if arg .lt.  fac
;.a= Ø, c=1 if arg .eq.  fac
;.a=-1, c=1 if arg .gt.  fac
;
dcomp     lda facexp
          eor argexp
;
;are mantissa signs same?
;
          lsr a
          bcs dcom1Ø  ;no...
;
;are exponent signs same?
;
          rol a
          bmi dcom2Ø  ;no...
;
;are exponent magnitudes same?
;
          bne dcom3Ø  ;no...
;
;compare mantissa magnitudes
;
          ldx #facmsd-faclsd-1
          sed
dcom5     lda arglsd+1,x
          cmp faclsd+1,x
          bcc dcom7
          bne dcom7
          dex
          bne dcom5
dcom7     cld
          bne dcom4Ø
          txa
          beq dcom45
;
;case different mantissa signs
;
dcom1Ø    lda facexp
          ror a
          jmp dcom42
;
;case different exponent signs
;
dcom2Ø    lda facexp
          rol a
          jmp dcom4Ø
;
;case different exponent magnitudes
;
dcom3Ø    sec
          lda argexp
```

```
          sbc facexp
;
;handle negative mantissa
;
dcom40    rol a
          eor argexp
          lsr a
;
;common exit code
;
dcom42    lda #$ff
          bcs dcom45
          lda #$01
;
dcom45    rts
.end
```

APPENDIX C

KEY TO KERNAL ERROR MESSAGES

Appendix c

ERROR CODES

Ø	Stop Key termination
1	Too many files
2	File open
3	File not open
4	File not found
5	Device not present
6	Not input file
7	Not output file
8	Missing file name
9	Illegal device number

APPENDIX D

SYSTEM RAM VECTORS

Appendix D

SYSTEM RAM VECTORS

relative address	name	function
Ø	IRQ	Hardware IRQ handler
2	BRK	Software interrupt handler
4	NMI	Hardware NMI handler
6	OPEN	Open file routine
8	CLOSE	Close file routine
A	CHKIN	Open channel for input
C	CHKOUT	Open channel for output
E	CLRCH	Clear channel
1Ø	CHRIN	Input from channel
12	CHROUT	Output to channel
14	STOP	Scan STOP key
16	GETIN	Get from channel
18	CLALL	Close all files
1A	USRCMD	Extend monitor commands