

THE SOURCE

by

The Pendle Connection

Published By Merlin Software

Contents

CHAPTER ONE

- Overview of the Basic Language
- Constants
- Variables
- Operators
- Expressions
- Functions
- Statements

CHAPTER TWO

- Number Systems
 - The Binary number system
- Programming Terminology
- Binary Arithmetic
 - Binary Addition
 - Binary Subtraction
 - Binary Division
- The Hexadecimal number system
- Hexadecimal addition
- Storing numbers in memory
- Negative values
- Two's complement numbers
- Logical operators

CHAPTER THREE

- Strings
 - String Descriptor Blocks
 - Ascii characters
 - String functions
 - Asc, left\$, right\$
 - mid\$, chr\$, len, scrn\$
 - str\$, val\$
 - Memory usage

CHAPTER FOUR

- Arrays
 - Data & Read statements
 - Subscripted variables
 - Dim statement
 - Two dimensional arrays
 - Using dimensioned arrays in programming games

CHAPTER FIVE

Three dimensional arrays

CHAPTER SIX

Using the POKE command
Rom & Ram
Memory organisation under Basic
System Variables
The Program Statement table
Keywords & Tokens
Structure of a Basic Line

CHAPTER SEVEN

Overview of the Z80 instruction set
The Z80 Registers
Addressing Modes
Comparison code Basic & Assembler

CHAPTER EIGHT

Writing Machine code routines
Adding a new command to Basic
Using Gen80 assembler for machine code subroutines
Using Pmon to install an new Basic command word
Storing subroutines in REM statements
Storing Graphics in strings

PART TWO

CHAPTER NINE

The video display processor
Reading and writing to Vram
The Vdp registers
Writing to the VDP registers

CHAPTER TEN

The different display modes
Pattern generator
Colour Table
Screen addressing

Pixel plotting
Setting up a Mode 1 screen

CHAPTER ELEVEN

Sprites
Setting up the Sprite attribute table
The Early clock bit
Locking off sprites
Sprite animation
Sprite collisions
Detecting sprite collisions
Detecting sprite collisions with the pattern plane

CHAPTER TWELVE

Interrupts and the CTC
Setting up Mode 2 interrupts
Using interrupts within a program

CHAPTER THIRTEEN

Sound
Overview of the Programmable Sound Generator
The Psg Registers
Using the Psg from machine code

CHAPTER FOURTEEN

Disc drives
Using CP/m protocol
Opening disc files
Closing disc files
Reading data from discs
Creating new disc files

APPENDIX

CONSTANTS

Constants are a set of values that never change. A constant can be a number or a literal string. **PI** is a constant. There are two types of numerical constants, **integer** or non-decimal figures like 1, 12, 6309, 25 and **real** constants like 33.3333 or 126.56E3. **Literal** constants take the form, "Keith", "Brian", "ABC", "Lxt42". The quotation marks are not part of the constant but are used as delimiters.

Scientific or Exponential notation [E] maybe unfamiliar to you but it is not to be avoided as the explanation is simple. E represents the number of positions left or right that the decimal point must be moved to produce a standard **Floating-point** number.

6.E4 can be written as $6.4 * 10^3$ or
 $6.4 * 1000 = 6400$

A negative power may be expressed in the form 5.75E-2 which is another way of writing .0575. Basic automatically stores floating-point numbers in scientific notation.

VARIABLES

Variables are exactly what the name implies - they are variable. Variables are simply names that the program uses to set aside memory locations for storing data. Albert's Basic allows you to use variable names of great length but the interpreter will only use the first five characters. **BLAGBOROUGH** will be stored as **BLAGB** and the rest of the characters will be ignored by Basic. Using variables of this length is really a waste of memory except for adding clarity to your program and such extravagance only eats up computer memory. Examples of variable names are **X**, **Y**, **A5**, **ALB**, **T566%**

Variables are automatically treated as floating point unless suffixed with the % sign. **TEST** will be treated as a real number and **TEST%** will be considered an integer.

You should be aware that Basic will accept a number in the range -65535 to 65535 the values returned will be in the range -32768 to +32767.

	X% = 65535	X% = 32767	X = 65536
	PRINT X%	PRINT X%	PRINT X
WILL BE DISPLAY	-1	32767	65536

Variable names are subject to the following rules:

- a) The first character must always be alphabetic.
- b) Any following characters may be alphanumeric but only the first five are considered by Basic.
- c) The next character may be % to denote an integer variable, otherwise the variable is treated as a real number. The \$ can also be used to denote a string variable.
- d) A variable name containing a reserved word is considered illegal and will not be accepted by the interpreter.

```
LBSET      IS A VALID VARIABLE
ABSET      IS NOT A VALID VARIABLE NAME
EVALUATION$ = "ABCD" IS INVALID
VALUATION$ = "ABC" IS ALSO INVALID
```

Wherever possible use integers: Basic stores these values in two bytes. The use of real variables when it is not necessary is a waste of memory and will eventually lead to your Basic program slowing down. For example, if the variable X will never exceed the value 32767 then use X%

Another type of variable used by Basic is the **SUBSCRIPTED VARIABLE**. This is a very powerful and important variable that will allow you to keep ordered lists, or store and access data in a random fashion.

A typical example of a subscripted variable is X(1) where X is the variable with a subscript of one. A further example is A(6) meaning a subscript of six. All subscripted variables that carry the same name e.g. A(1), A(2), A(3).....A(13) etc. constitute an array.

Arrays are created by DIMensioning them at the start of your program - this is not strictly true but it is good programming practice to do so - DIM AL(12) will dimension an array AL(0) through to AL(12) giving an array of thirteen elements.

If an array is never going to exceed eleven elements there is no need to use the DIM statement as Basic automatically sets a default value of ten to any array that has not been dimensioned with the DIM statement.

The subject of arrays is too important to dismiss in a few short lines and we will discuss the matter at length in a later chapter.

OPERATORS

When a string or arithmetic expression is evaluated the result depends upon

the priority of each operator and the presence or absence of parentheses. Parentheses take the highest precedence and insure, in both arithmetic and string manipulation, that the expression enclosed within the parentheses is evaluated before any other calculations are carried out. The rules for parentheses are the same as we learned at school, and any expression in brackets within another set of brackets will be evaluated first.

$$\frac{2+(3*4)}{2+5}$$

Ans = 2

()	PARENTHESIS	HIGHEST PRIORITY
^	EXPONENTS		
* / MOD	MULTIPLY OR DIVIDE OR MODULOUS		
+ OR -	ADD OR SUBTRACT		
< = OR >	COMPARISONS LESS THAN, EQUAL TO, GREATER THAN		
NOT	LOGICAL NOT		
AND	LOGICAL AND		
XOR OR	LOGICAL EXCLUSIVE OR, LOGICAL OR		

String comparison is not quite so straight forward as arithmetical operations. Strings are compared in alphabetic type sorting where $P < Q$. The comparison is carried out character by character until the end of the shorter string is encountered, or until a position is reached where the two strings differ. The string with the greatest value is the string that contains the character with the highest ASCII value.

With strings of unequal length the shorter string is less than the longer string: "LOCATE" < "LOCATED". When comparing two or more strings Basic compares the Ascii value of each character in the string: AB > AA and AC < AX. Also note that "a" is greater than "A". [see Ascii table]

Type in the following short program and you will soon get the drift of how Ascii codes are used in string comparison.

```

10  CLS
20  X$ = INCH$
30  PRINT@ 2,5;X$;"=";"ASCII CODE";ASC(X$)
40  GOTO 20

```

EXPRESSIONS

There are four classes of expression used with Basic: numeric, string, relational and logical.

A numeric expression may consist of numbers, variables (real or integer), operators, logical expressions and parentheses.

```
100  A = ABS(M) + C * COS(N)
```

A string expression may be an arrangement of string functions, string variables, string literals, parentheses and the string operator for concatenation +.

```
100  H$ = CHR$(10)+H$+LEFT$(Y$)+"IAN"
```

Relational expressions compare two numerical or string values.

```
100  IF X$ <= Y$ THEN GOTO 40
```

The result of evaluating a logical expression are:

-1 FOR TRUE
0 FOR FALSE

```
100  IF X = 30 AND Y = 40 THEN PRINT "YES"
```

IF G THEN GOTO is another expression that will ensure whenever $G \neq 0$ the program will always jump to line 100. IF G AND 16 is also allowed. It should be obvious that the correct use of logical expressions can save a lot of typing. However, care should be taken when structuring your expressions. Take a look at the following:

```
100  IF T = 0 AND Y = 0 THEN GOTO 55
```

In the above expression if T and Y are equal to zero the program will jump to line 55. Now. Consider this next expression:

```
100  IF T AND Y = 0 THEN GOTO 55
```

In this expression the program will **always** branch to line 55 whenever $T \neq 0$ and $Y=0$ which is not the same as the previous example. We shall return to logical functions a little later in the book.

FUNCTIONS

Functions can be user definable using the DEF FN statement or they can be

THE SOURCE CHAPTER ONE

part of the Basic language.

User defined functions can be thought of as subroutines that take a number of parameters and return a value depending on the complexity of the statement.

Basic functions normally require an expression to be enclosed in brackets. ABS(X) is a function. Numerical functions return numeric values and string functions return string values.

SOME EXAMPLES OF FUNCTIONS:

```
COS (n)
LOG (n)
LEFT$ (Y$)
```

STATEMENTS

Basic programs are composed of Basic Lines. Basic lines are made up from Basic Statements. A statement is a command that tells the computer to carry out some specific action and can fall into one of the following categories:

Assign statements:

LET VARIABLE = EXPRESSION

LET X = Y + 256/4

Of course, the LET is optional on the Einstein.

Control Statements

GOSUB RETURN GOTO

Loop Statements

FOR NEXT LOOPS

Conditional Statements

IF LOGICAL EXPRESSION THEN STATEMENT

REM statements

Allow comments to be inserted within programs in order to aid clarity when program is used by other programmers.

Output Statements

Used to send data to another device : Vdp, printer or disc

Input statements

Used to retrieve data from an external device such as keyboard, vdp or disc drives.

In the
hexadecimal
skip the
want to
memory
time
your file

BINARY NUMBERS

A memory location is called a "cell". This integer value is the location stored in memory that are used by your program. Though you can use decimal numbers within computer code, all binary numbers - all hexadecimal numbers - are converted (internally) into binary and stored in the memory.

These binary numbers are known as "machine code". These instructions are usually a set of binary bits in order and represent a value of ON (1) or OFF (0). The ON/OFF patterns and acts accordingly. If you use hexadecimal and binary numbers interact you will find efficient programs and performing operations mentioned in the manual e.g. bit testing or set, etc.

CHAPTER TWO

In this chapter we are going to consider the subject of **decimal, binary and hexadecimal numbers** and how they are stored in computer memory. No! Don't skip this section - if you are to advance into assembly language or you want to Peek & Poke around in memory you must understand the layout of memory. Anyway, it's easier to understand than you may think and a little time spent learning the ground rules will be rewarded a hundred fold in your future attempts at programming.

BINARY NUMBERS

A memory location in the Einstein has an "address". This address can be integer from 0 through to 65535. Memory locations store data and commands that are to be used by the computer to **RUN** your program. However, even though you may be dealing with decimal numbers within your program, the computer can only operate on binary numbers - all your decimal and hexadecimal numbers are converted (internally) into binary before being stored in the computer's memory.

These binary numbers are known as machine code instructions. Each of these instructions is actually a set of binary bits arranged in a certain order and represent a state of ON [1] or OFF [0]. The computer recognises the ON/OFF patterns and acts accordingly. If you understand how decimal, hexadecimal and binary numbers interact you will be capable of writing more efficient programs and performing operations from Basic not normally mentioned in the manual e.g. bit testing or compacting data, to name but two.

THE SOURCE CHAPTER TWO

One memory location can store 8-binary digits. One binary digit is called a bit and eight bits make up one byte.

ONE MEMORY LOCATION = 8 BITS = 1 BYTE

When counting in decimal we use the digits 0 to 9. To carry on counting after 9 we must go back to 0 and carry a 1 into the "tens" column, and so on thereafter.

We know that $152 = (1*100) + (5*10) + (2*1)$ and we can also write 152 in its expanded form by expressing it as $1 * 10^2 + 5 * 10^1 + 2 * 10^0$ - think back to your school days ANY NUMBER RAISED TO THE POWER OF ZERO $[10^0] = 1$.

The binary system uses only digits 0 & 1. When we count in binary the same rules apply as in the decimal system but this time after counting to 1 we go back to 0 and carry 1 into the next column left. The binary number 0111 can be written as:

$$1*2^2 + 1*2^1 + 1*2^0 \text{ or } [2 * 2] + [2 * 1] + 1 = 7$$

In binary, each position represents a power of two rather than the power of ten.

$$1011 = 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 11$$

The eight binary digits [bits] are numbered 0 to 7 from right to left. If we now examine the notation of one byte you will see how easy it is to calculate the equivalent decimal number using this positional notation.

Positional notation ==>	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	+-----+							
Binary number ----->	1	1	0	0	0	1	1	1
	+-----+							
Bit Number ----->	7	6	5	4	3	2	1	0

The above binary number in decimal is :

$$\begin{aligned}
 &1*2^7 + 1*2^6 + 0*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 1*2^0 \\
 &= (2*2*7) + (2*2*6) + (2*2) + 2 + 1 \\
 &= 128 + 64 + 4 + 2 + 1 = 199
 \end{aligned}$$

You can see quite clearly from the above example that as you move to the next bit position left the previous value doubles.

Positional value	----->	128	64	32	16	08	04	02	01
Bit number	----->	7	6	5	4	3	2	1	0

If all the positional values are added together we find that one byte can hold a maximum value of 255 or 1111 1111 binary.

Positional notation can be extended to two or more bytes. When two bytes [16-bits] are used in this way, much larger numbers can be represented.

TABLE 2.1
TABLE OF VALUES FOR TWO BYTE BINARY NOTATION

DECIMAL	+	BINARY															+
		15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	+	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	+	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
254	+	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0
255	+	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1024	+	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
24576	+	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
24577	+	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1
32000	+	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0
32767	+	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

You will notice in Table 2.1 we only use 15 bits which allows the a maximum value of 32767 to be stored in two bytes - the reason for this anomaly will be discussed a little later in this chapter. This method of storing numbers is known as integer format. With integers you are not allowed to use numbers greater than 32767. Test it for yourself by typing in the following short program.

```

10  CLS
20  INPUT "INPUT ANY NUMBER ";I%
30  PRINT I%
40  GOTO 20

```

Run the program and try answering the prompt with different values for I%.

THE SOURCE CHAPTER TWO

Then try giving I% a value greater than 327627. What happened? The computer returned the **two's complement** of the number which is how Einstein's Basic differs from other Basics - they would give an Overflow Error. Now type in a number greater than 65535. This time the computer responds with QTY ERROR IN 20 meaning you tried to assign a value to the integer variable I% that was out of range.

TABLE 2.2
POWERS OF TWO

=====			
2^0	=	1	2^8 = 256
2^1	=	2	2^9 = 512
2^2	=	4	2^{10} = 1024
2^3	=	8	2^{11} = 2048
2^4	=	16	2^{12} = 4096
2^5	=	32	2^{13} = 8192
2^6	=	64	2^{14} = 16384
2^7	=	128	2^{15} = 32768
=====			

TERMINOLOGY

It was once fashionable to compare binary digits with light bulbs or electric switches where 1 represented the switch being set to the **on** position and 0 represented the switch being set to the **off** position. This terminology is still used today: if a bit is 1 we say it is **set**, if a bit is 0 we say it is **reset**.

We shall now be dealing with three number systems - decimal, hexadecimal and binary. To distinguish between the three systems the following will be used: H = hexadecimal otherwise the numbers is decimal. Binary is visually of that number system.

BINARY ARITHMETIC

Binary addition and multiplication are easy operations to understand. As there are only two digits [0 & 1] to deal with in binary, we only have four permutations to learn for addition and four for multiplication.

$$+ \frac{0}{0} + \frac{0}{1} + \frac{1}{0} + \frac{1}{1} = 10$$

$$* \frac{0}{0} * \frac{0}{1} * \frac{1}{0} * \frac{1}{1} = 1$$

From the above you can see that the only rule to remember is:

ADDITION

$$1 + 1 = 0 \text{ and carry } 1$$

Carry Line 1 1 1 1

	15	=	1 1 1 1
	06	=	0 1 1 0
Answer	21	=	1 0 1 0 1

Method:

0 + 1 = 1 one goes in the answer. 1 + 1 = 10 put 0 in answer and carry 1.
1 + 1 = 10 + carry 1 = 11 put 1 in answer and carry 1. Put carry in answer line.

BINARY ADDITION TABLE

=====			
:	+	:	:
0	:	0	1
1	:	1	0
=====			
:	:	:	:
0	:	0	0
1	:	1	10
=====			

MULTIPLICATION

Multiplication only consists of adding a value to itself a given number of times: $4 * 3 = 4 + 4 + 4 = 12$: $6 * 5 = 6 + 6 + 6 + 6 + 6 = 30$.

Let's now take a look at some binary multiplication and you will see that a couple of interesting facts emerge.

BINARY MULTIPLICATION TABLE

=====				
:	*	:	0	1

:	0	:	0	0
:	:	:	:	:
:	1	:	0	1
=====				

Fact One: Whenever a 1 appears in the multiplier, the multiplicand is copied into the partial answer column. If 0 appears in the multiplier, the multiplicand is not copied.

Fact Two: On each step the partial answer is shifted one place to the left even if the multiplier contains zero. This fact provides us with a quick method of multiplying binary numbers for powers of two: 2, 4, 8, 16 etc.

$2 * 2 = 4 \dots 2 = 0000\ 0010$ now shift left one place = $0000\ 0100 = 4$.

$2 * 8 = 16 = 2 * 2 * 2$
 $2 = 0000\ 0010$
 shift left two places
 $= 0000\ 1000 = 8$

Multiplicand	1 0 1 0	= 10
Multiplier	0 1 1 0	= 06
	0 0 0 0 partial answer
	1 0 1 0 "
	1 0 1 0 "
	0 0 0 0 "
Product	0 1 1 1 1 0 0	= <u>60</u>

BINARY DIVISION

Binary division is simplicity itself because you can see at a glance if one number will divide into another. Division is the inverse of multiplication and can, therefore, also be performed by successive subtraction until a negative remainder is encountered.

If division is the inverse of subtraction can we inverse fact two when dividing by multiples of two? Yes! We can. When dividing by multiples of two shift one place to the right.

Divide 8 by 4.

4 = 2 * 2 so we shift **two** places right.
 8 = 0000 1000
 Shift one place right = 0000 0100 = 4
 Shift one place right = 0000 0010 = 2

Long Binary Division.

		1 0 1 1 quotient
Divisor	1 0 1	1 1 0 1 1 1 dividend
		<u>1 0 1</u>	
		1 1 1	
		<u>1 0 1</u>	
		1 0 1	
		<u>1 0 1</u>	
		0 0 0 remainder

HEXADECIMAL NUMBERS

Once you have learned the rudiments of the binary system, hexadecimal numbers are not difficult. In hexadecimal notation we need the digits 0-15. Normal conventions substitute A B C D E F for decimal digits 10 through to 15. Using this convention the decimal number 13 becomes 0D hexadecimal or 0DH.

THE SOURCE CHAPTER TWO

Hexadecimal numbers allow us to manipulate binary numbers in a more manageable fashion. Large binary numbers are cumbersome and look very similar to each other and spotting the difference between 11110101 00101110 and 1110101 00101010 in a list of binary numbers is not easy. Consider the start of Albert's Scratch-pad locations. In decimal the address is 64256 and in binary this becomes 11111011 00000000. The hexadecimal address is far easier to remember FB00H.

Two hex (short for hexadecimal) digits make up one byte [8 bits] and one hex digit makes up four bits (often called a nybble). By splitting our binary numbers into groups of four it is relatively simple to convert them into hexadecimal notations.

TABLE 2.3
HEXADECIMAL CODING

BINARY	:	DECIMAL	:	HEX
0000	:	0	:	0
0001	:	1	:	1
0010	:	2	:	2
0011	:	3	:	3
0100	:	4	:	4
0101	:	5	:	5
0110	:	6	:	6
0111	:	7	:	7
1000	:	8	:	8
1001	:	9	:	9
1010	:	10	:	A
1011	:	11	:	B
1100	:	12	:	C
1101	:	13	:	D
1110	:	14	:	E
1111	:	15	:	F

Let's now take a look at the binary number 1110 1100 0100 1101 = 60493 decimal. If you look at the hexadecimal values in table 2.3 you can see how to convert the value into a hex number by equating each set of four bits to the equivalent hexadecimal digit.

$$\begin{array}{cccc}
 \text{E} & \text{C} & 4 & \text{D} \\
 1110 & 1100 & 0100 & 1101 \\
 = (16^3 * 14) + (16^2 * 12) + (16^1 * 4) + 13 \\
 = 60493 \text{ Decimal.}
 \end{array}$$

HEXADECIMAL ADDITION

Addition using hexadecimal numbers is very similar to adding decimal numbers except that we are counting to 16 before carrying a 1 into the next column left.

```

100 .... carry line .... 1110
AC01H                               DACBH
1FC3H                               2BB0H
CBC4H .. answer line .. 1067BH

```

Table 2.4 will assist in hexadecimal additions. If you want to add 0CH to 0BH look up C in the left-hand column and move over the columns of figures until you find the intersection with B in the top column, you will see that the answer is 17H. If you were actually adding these two numbers you would put 7 in the answer line and 1 in the carry line.

TABLE 2.4
HEXIDEIMAL ADDITION

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

* remember 10 11 etc. refer to 10 hex, 11 hex = 16, 17 decimal *

Let us now consider how the Einstein actually stores numbers in its memory.

All Z80 based computers store their numbers in what is termed Least significant byte [LSB] / Most significant byte format [MSB]. In simple terms; they store their numbers back to front ! The reason for this is not obvious but take my word, using this method with very large numbers results in memory saving of more than sixty percent.

THE SOURCE CHAPTER TWO

If we wanted to store the decimal number 60493 in memory location 65450 we first have to convert it into two values: LSB & MSB. Once these values have been calculated we put the Lsb into 65450 and the Msb into 65451. To find the Lsb and Msb values we can use the following, easy to learn, formula :-

$$\begin{aligned}\text{MSB} &= \text{INT}(\text{NUMBER}/256) \\ \text{LSB} &= \text{NUMBER} - (256 * \text{MSB})\end{aligned}$$

Using the above formula we can now convert our decimal number 60493.

$$\begin{aligned}\text{MSB} &= \text{INT}(60493/256) = 236 \text{ decimal} = \text{OECB} \\ \text{LSB} &= 60493 - (256 * \text{MSB}) = 77 \text{ decimal} = \text{04DH}\end{aligned}$$

If we were programming in Basic we could put this number into memory using the POKE command.

```
POKE 65450,LSB
POKE 65451,MSB
```

It is important to understand this rule. Later, when we start POKEing machine code instructions into memory, we must know how to store and retrieve them correctly.

Einstein Basic allows an easier way of putting data into memory and this is using the **DOKE** command which allows us to poke the number into memory without splitting it into Lsb,Msb - the DOKE command does this automatically.

```
DOKE 64450,60493
```

This single command will store the number exactly as we discussed in the previous paragraphs. However, it is still important to know how the POKE command operates as most computers use this convention, and there are times when it is desirable to alter the Msb or Lsb of a number already stored in memory.

How a two byte number is stored in memory:-

Bit Positions ----->	7	6	5	4	3	2	1	0	
Address 65450	: 0	: 1	: 0	: 0	: 1	: 1	: 0	: 1	: Lsb
Address 65451	: 1	: 1	: 1	: 0	: 1	: 1	: 0	: 0	: Msb

NEGATIVE VALUES

Up to this point we have been discussing eight & sixteen-bit integers. These are referred to as unsigned integers. However, there is a lot more to computing than manipulating this type of numeric data - what about negative values? A computer wouldn't be much of an asset if it would not perform normal mathematical functions. Clearly, to work with positive and negative data, we need some method of distinguishing a positive integer from a negative value. To do this, we have to abide by a set of rules that is used in the majority of digital computers.

In a **signed 8-bit integer** bit-7 (Most Significant Bit) is treated as the **sign bit**. IF BIT 7 IS SET [1] THEN THE NUMBER IS **NEGATIVE**. IF BIT 7 IS RESET [0] THEN THE NUMBER IS A POSITIVE VALUE or at least equal to zero.

0100 1111 = 79 decimal 1100 1111 = -79 decimal

Whenever we use unsigned integers the largest number we can work with in one byte is 255 decimal or 1111 1111 binary, and the smallest is zero. The largest number we can represent as a signed 8-bit integer is 127 decimal or 0111 111 binary, and the smallest value is -127 decimal, 1111 1111 binary.

Using this method of representing 8-bit values soon poses a problem: **we cannot use normal conventions for adding two signed numbers**. Study the following example :-

Add 79 & -13 together in binary :

0100 1111	=	79	decimal
+ 1000 1101	=	-13	decimal
1101 1100	=	-192	decimal

The answer is [-92] is obviously wrong ! It should be +66 decimal.

TWO'S COMPLEMENT REPRESENTATION

To overcome this anomaly, some brilliant brain - no not me, in the distant past, discovered the rule of two's complement. There is nothing mysterious about two's complement representation and it is quite simple to calculate.

THE SOURCE CHAPTER TWO

Rule (a)

The two's complement of a positive 8-bit binary number is the number itself.

Rule (b)

The two's complement of a negative 8-bit value is calculated by obtaining the one's complement of the positive value and adding one.

Calculating the complement of a number is just a matter of changing all the 1's to 0's and all the 0's to 1's.

Find the two's complement of 16 decimal.

16	=	0001 0000	
		1110 1111 one's complement
		1 add 1
		1111 0000 two's complement

Find the two's complement of 102 decimal

102	=	0110 0110	
		1001 1001 one's complement
		1 add 1
		1001 1010 two's complement

If we use the logical operator **NOT** we can vary rule (b) to read :-

NOT the number (ignoring the sign) then add 1

If we now use our original example and add +79 to -13 decimal we will see that this method works correctly.

NOT 13	=	1111 0010	
		1 add 1
		= 1111 0011 two's complement

		0100 1111 +79 decimal
		+ 1111 0011 -13 two's complement
		(1)0100 0010	= <u>+66 decimal</u>

+66 is the correct answer !

Add -3 to -2 decimal

3 = 0000 0011 complement = 1111 1100 ... add 1 = 1111 1101
2 = 0000 0010 complement = 1111 1101 ... add 1 = 1111 1110

```

      1111 1101
      1111 1110
(1) 1111 1011  = -5 decimal

```

The carry from bit-7 is ignored completely and our result is :- 1111 1011 which should be the two's complement of -5. We can test this quite easily by working in reverse ...

```

NOT 1111 1011 = 0000 0100
      1      add 1
0000 0101 = 5 decimal

```

Therefore 1111 1011 is the correct result for adding the two negative numbers.

The range of numbers that can be stored in an 8-bit two's complement representation is +127 decimal down to -128 decimal. [see Table 2.5]

Although we have only discussed 8-bit integers everything we have said also applies to 16-bit or 32-bit values. The number range for unsigned 16-bit integers is 0 - 65535 decimal and for 16-bit two's complement representation -32768 through to +32767. You should now realise why bit 15 was left unused in Table 2.1, it was to allow for the sign bit.

TABLE 2.5
EXAMPLE OF TWO'S COMPLEMENT CODES

=====			
: + :	2'S COMPLEMENT :	: - :	2'S COMPLEMENT :
=====			
: 127 :	0111 1111	:-128 :	1000 0000
: 126 :	0111 1110	:-127 :	1000 0001
:	:	:	:
: 64 :	0100 0000	:- 64 :	1100 0000
: 63 :	0011 1111	:- 63 :	1100 0001
: 32 :	0010 0000	:- 32 :	1110 0000
: 31 :	0001 1111	:- 31 :	1110 0001
:	:	:	:
: 16 :	0001 0000	:- 16 :	1111 0000
: 15 :	0000 1111	:- 15 :	1111 0001
: 1 :	0000 0001	:- 1 :	1111 1111
=====			

LOGICAL OPERATIONS

An often neglected feature of the computer is its ability to perform logical operations. The Einstein's Basic instruction set is very powerful and provides a worthy set of logical operations that can be called from a Basic program.^o

In Boolean Algebra there can be only two answers or states : TRUE [-1] & FALSE [0]. If the result of a logical operation is true the computer sets all the bits of the byte to ones, on the other hand, if the result is false the eight bits are reset to zero.

Logical operations are not difficult to comprehend, after all, we use the AND statement recurrently within our Basic programs.

```
10 IF X = 3 AND Y = 1 THEN GOTO 30 : ELSE GOTO 100
```

Every time we program the computer with IF, THEN, ELSE statement we are actually using logical expressions.

In the previous example if X = 3 and y = 1 then the condition is true and the program will jump to line 30. Any other values in X and Y will result in the program branching to line 100.

Logical expressions in the form : IF G THEN GOTO 100 are also allowed in Einstein Basic. Whenever G <> 0 the program will always jump to line 100. IF G AND 16 is also allowed - this is very useful shorthand will say a lot of typing.

The use of AND, OR, NOT isn't restricted to simple relational expressions as mentioned above. These three operators can also be used for Boolean operations, bit manipulation and bit comparison.

NOT

The NOT expression forms the complement - as already discussed - of the number by inverting each bit within the byte.

```
NOT 12 ---> -13
NOT -2 ---> 1
NOT 0 ---> -1
```

```
NOT 0 .... 0000 0000
           1111 1111 = -1
```

AND

The AND operation is used to mask out certain, unwanted, bits from a byte.

12 AND 14 ----> = 4

AND 0000 1100 = 12 decimal
 0000 0100 = 4 decimal
 0000 0100 = 4 decimal

25 AND 12 ----> = 8

AND 0001 1001 = 8 decimal
 0000 1100 = 12 decimal
 0000 1000 = 8 decimal

4 AND 2 ----> = 0

AND 0000 0100 = 4 decimal
 0000 0010 = 2 decimal
 0000 0000 = 0 decimal

You can test the above for yourself using Basic. Just type in the following then press ENTER.

PRINT 4 AND 2

XOR (eXclusive OR)

XOR is often used to set a byte to zero, it can also be used for plotting to the Vdu. it is also very useful in animation sequences. Any value Xor'ed with itself produces zero.

3 XOR 3 ----> = 0

4 XOR 2 ----> = 6

OR 0000 0100 = 4 decimal
 0000 0010 = 2 decimal
 0000 0110 = 6 decimal

In a later chapter we will see how this logical operation can be used as a "switch" to manage animation sequences. Logical operators should never be ignored they are very useful and in assembly language programming they are indispensable.

THE SOURCE

CHAPTER TWO

TABLE 2.6
LOGICAL OPERATIONS ON BITS

NOT :	0	1
	1	0

AND :	0	1
0 :	0	0
1 :	0	1

OR :	0	1
0 :	0	1
1 :	1	1

XOR :	0	1
0 :	0	1
1 :	1	0

TABLE 2.7
LOGICAL OPERATIONS TRUTH TABLE

=====				
NOT	TRUE	=	FALSE	
NOT	FALSE	=	TRUE	
TRUE	AND	TRUE	=	TRUE
TRUE	AND	FALSE	=	FALSE
FALSE	AND	TRUE	=	FALSE
FALSE	AND	FALSE	=	FALSE
TRUE	OR	TRUE	=	TRUE
TRUE	OR	FALSE	=	TRUE
FALSE	OR	TRUE	=	TRUE
FALSE	OR	FALSE	=	FALSE
TRUE	XOR	TRUE	=	FALSE
TRUE	XOR	FALSE	=	TRUE
FALSE	XOR	TRUE	=	TRUE
FALSE	XOR	FALSE	=	FALSE
=====				

We have covered quite a lot of ground in this chapter so don't worry if you haven't fully understood what has been discussed. You can always refer back to this at a later stage, and a lot will "sink in", automatically, as you progress through the rest of the book. Usage is the only way to become familiar with number systems.

CHAPTER THREE

The Einstein doesn't only manipulate numerical values. It is a master at juggling with letters and text. **Strings** are simply lengths of data that can be constructed from numbers, letters, special control characters and graphic codes.

Normally, strings are created from Ascii characters. Ascii stands for American Standard Code for Information Interchange. This is a standard that has been adopted by most of the computer industry to do exactly what it says - allow micros to interchange printable characters in a standard format e.g. when using the RS232 to communicate with a computer or bulletin board.

Ascii is coded into 7-bits (bits 0 - 6) and we have already learned, in the previous chapter, that 7-bits can hold a number in the range 0 - 127. This suggests that 128 different codes can be defined from one byte and these characters form the standard Ascii character set. Graphic characters are allowed by setting bit 7 which allows a further 128 characters to be defined chr\$(128) through to chr\$(255), however, these will not necessarily be recognised by another computer.

Although we have said that characters with Ascii codes 0 - 127 form the standard codes, this is not strictly true. Certainly, the displayable codes 32 (SPACE) up to 122 (z) are standard, but the codes 0 - 31 known as control codes may differ from computer to computer. [see Table 3.1]

String variables do not store their data in the actual Basic program line. As they are created the string name is stored as a string descriptor block in the simple variable table and the physical construction, or content of the string, is copied into the string work area (see diagram 3.1).

THE SOURCE CHAPTER THREE

Strings can be up to 255 characters in length and the amount of memory the string occupies is totally dependant on its physical length - one byte per character. The length of the string name does not affect memory usage as there are only four bytes allocated, in memory, for this purpose.

DIAGRAM 3.3
STRING DESCRIPTOR BLOCK

```
+-----+-----+-----+-----+
+ String Name : Length : Lsb Address : Msb Address :
+-----+-----+-----+-----+
```

A\$ = "STRING" is stored as follows :-

```
+-----+-----+-----+-----+
+ 00 : 00 : 00 : 01 : 06 00 : E0 : 8A :
+-----+-----+-----+-----+
  Δ      Δ      Δ      Δ      Δ      Δ      Δ      Δ
  :.....:.....:.....:      : Ignored :      :
  :      :      :      :      : .....:
  String name bit String
  zero set to den- Length
  ote string.     Pointer to address
                   in String Work Area
```

The String Descriptor Block stores the string name in a rather complicated manner and the first four bytes contain the name with bit zero of byte four being set to denote that the variable is a string. The following bytes contain the string length and the actual memory address where the physical attributes of the string are stored.

String variables cannot be operated on in the same manner as numerical variables - after all, who would want to use an expression :-

```
100 A$ = "Albert" : B$ = "Einstein" : IF A$/B$ = "HELLO" THEN GOTO 200
```

This sort of statement is meaningless when applied to a string. Strings can, however, use the + (plus sign). Two or more strings can be linked together with the + sign and this operation is termed **concatenation**.

```
10 W$ = "HELLO" : X$ = " I AM YOUR "
20 Y$ = "EINSTEIN COMPUTER."
30 PR$ =W$+X$+Y$
40 PRINT PR$
```

The above program would display : HELLO I AM YOUR EINSTEIN COMPUTER.

TABLE 3.1
ASCII CHARACTER CODES.

* CODE	CHARACTER	* CODE	CHARACTER	* CODE	CHARACTER	*
* 0		* 43	+	* 86	V	*
* 1	SCREEN DUMP	* 44	,	* 87	W	*
* 2		* 45	-	* 88	X	*
* 3		* 46	.	* 89	Y	*
* 4	CURSOR RIGHT	* 47	/	* 90	Z	*
* 5		* 48	0	* 91	[*
* 6	Delete at cursor	* 49	1	* 92	\	*
* 7	BEEP	* 50	2	* 93]	*
* 8	CURSOR LEFT	* 51	3	* 94		*
* 9	TAB	* 52	4	* 95		*
* 10	CURSOR DOWN	* 53	5	* 96	-	*
* 11	CURSOR UP	* 54	6	* 97	a	*
* 12	HOME & CLS	* 55	7	* 98	b	*
* 13	ENTER	* 56	8	* 99	c	*
* 14	CLS 40 COL	* 57	9	* 100	d	*
* 15	CLS 32 COL	* 58	:	* 101	e	*
* 16	CLS 80 COL	* 59	;	* 102	f	*
* 17	CURSOR ON	* 60	<	* 103	g	*
* 18	PRINTER ON	* 61	=	* 104	h	*
* 19	PRINTER OFF	* 62	>	* 105	i	*
* 20	CURSOR OFF	* 63	?	* 106	j	*
* 21	E.O.LINE	* 64	@	* 107	k	*
* 22	E.O.SCREEN	* 65	A	* 108	l	*
* 23	INVERSE/NORMAL	* 66	B	* 109	m	*
* 24	ERASE LINE	* 67	C	* 110	n	*
* 25	DELETE LEFT	* 68	D	* 111	o	*
* 26	INSERT	* 69	E	* 112	p	*
* 27	ESC	* 70	F	* 113	q	*
* 28	HOME CURSOR	* 71	G	* 114	r	*
* 29		* 72	H	* 115	s	*
* 30		* 73	I	* 116	t	*
* 31		* 74	J	* 117	u	*
* 32	SPACE	* 75	K	* 118	v	*
* 33	!	* 76	L	* 119	w	*
* 34	"	* 77	M	* 120	x	*
* 35	£	* 78	N	* 121	y	*
* 36	\$	* 79	O	* 122	z	*
* 37	%	* 80	P	* 123		*
* 38	&	* 81	Q	* 124		*
* 39	'	* 82	R	* 125		*
* 40	(* 83	S	* 126		*
* 41)	* 84	T	* 127		*
* 42	*	* 85	U	* 128		*

THE SOURCE CHAPTER THREE

Strings may be compared in the same manner as numerical variables are compared, and the operators employed are exactly the same as those used with numbers.

```
< Less Than
> Greater Than
<> Not Equal
<= Less Than or Equal to
>= Greater than or equal to
= Equal to
```

How strings are compared with each other lies in the use of Ascii codes. If you try the following program you should get the drift.

```
10 CLS
20 AN$ = INCH$
30 PRINT ASC(AN$); " ASCII CODE"
40 GOTO 20
```

ASC

The ASC function returns the Ascii value of the string e.g.

```
10 A$ = "A" : PRINT ASC(A$) ... will return 65
```

```
10 PRINT ASC("z") ... returns 122
```

When comparing two or more strings Basic compares the Ascii value of each character within the string. AB\$ will be greater than AA\$ and AC\$ will be less than AZ\$. You should also note that "a" is greater than "A". (see table 3.1)

When strings are of unequal length the shorter string is the less than the longer string. "LOCATE" < "LOCATED".

Earlier, we saw how strings could be concatenated by using the + sign, we are not, however, allowed to truncate by using the - (minus sign). We have to truncate indirectly by using LEFT\$, RIGHT\$, MID\$ functions. These particular functions allow us to access part or the whole of a string, starting in the middle, from the left, or from the right.

LEFT\$

```
10 A$ = "EINSTEIN"
20 B$ = LEFT$(A$,3)
30 PRINT B$
```

RETURNS B\$ = "EIN"

RIGHT\$

This function returns the last n characters starting from the right of the string variable specified.

```
10  A$ = "EINSTEIN"  
20  B$ = RIGHT$(A$,4)  
30  PRINT B$
```

RETURNS B\$ = "TEIN"

MID\$

MID\$ is used to take part of a string of length n, starting at position p within the specified string.

```
10  A$ = "THIS DEMONSTRATES THE MID$ FUNCTION"  
20  n = 6 : p = 12  
30  B$ = MID$(A$,n,p)  
40  PRINT B$
```

RETURNS B\$ = "DEMONSTRATES"

The MID\$ function can also be used on the right side of an argument to modify a names string.

```
10  A$ = "NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THE PARTY"  
20  B$ = MID$(A$,21,12)  
30  PRINT B$
```

RETURNS B\$ = "ALL GOOD MEN"

CHR\$

In an earlier part of this chapter we used the ASC function to convert from a character to its equivalent Ascii code. CHR\$ will allow us to convert in the opposite direction - it converts from Ascii code to a character.

CHR\$ permits us to use unprintable characters within our programs. Used in the correct manner it is an extremely powerful function.

"Why do we want to print the unprintable?" I hear you ask. Where is your sense of adventure?

If you look at Table 3.1 you will see that some of the codes are used to move the cursor in differing directions on the screen. How do we print the sound of the bell? With CHR\$(7) of course!

THE SOURCE

CHAPTER THREE

```
10 CLS
20 PRINT "XXXXXXXXXXXXXXXXXXXXXXXXXXXX";
30 PRINT CHR$(7)
40 FOR I = 1 TO 100 : NEXT
50 GOTO 30
```

See what I mean ? You have just printed an unprintable character. CHR\$ will allow us to do some clever things within our programs. Line 20 is there to demonstrate that although the program is sounding the bell, it is printing to the screen.

CHR\$ can also be used to compare single character strings :-

```
10 X$ = "A"
20 INPUT A$
30 IF A$ = CHR$(65) THEN PRINT "YES!":ELSE PRINT "NO!"
40 GOTO 10
```

You can use it in tandem with another function. Try working out the following program.

```
10 X$ = "A"
20 INPUT A$
30 IF A$ = CHR$(ASC(X$)) THEN PRINT "YES!" : ELSE PRINT "NO!"
40 GOTO 20
```

It does exactly the same as the previous program !

LEN

The LEN function will return the character length of a string and is a very useful function to have around when working with strings.

```
10 X = 0
20 INPUT A$
30 FOR I = 1 TO LEN(A$)
40 X = X+1
50 NEXT
60 PRINT "A$ IS ";X;" CHARACTERS IN LENGTH."
70 GOTO 10
```

HEX\$ & BIN\$ allow conversion from one number system to another and could be used within a program that converts number bases.

```
10 INPUT X
20 PRINT "BINARY IS ";BIN$(X)
30 PRINT "HEXADECIMAL IS ";HEX$(X)
40 GOTO 10
```


SCRN\$

This function is totally unrelated to the other string functions. SCRNS\$ allows us to grab any 40 column line that is displayed on the screen and store it within a named string variable.

```
10 LET TEMP$ = SCRNS$(23)
```

The above statement will store the last line of the screen display in a string variable TEMP\$. This can be re-displayed at any time by the statement : PRINT TEMP\$.

Finally, we come to two really powerful string functions :- VAL\$ & STR\$

VAL converts a string variable, or expression, into a numerical expression represented by the characters in the string argument - in fact, it eVALuates the string. The string must be numerical, and if it is a **real** number it must contain a decimal point or **exponent** [E].

B\$ = "100.60"	: PRINT VAL(B\$)	----->	RETURNS 100.60
B\$ = "999999"	: PRINT VAL(B\$)	----->	RETURNS 999999
B\$ = "9999999"	: PRINT VAL(B\$)	----->	RETURNS 1E+07
B\$ = "ABC45"	: PRINT VAL(B\$)	----->	RETURNS 0
B\$ = "45ABC"	: PRINT VAL(B\$)	----->	RETURNS 45
B\$ = "1E-3"	: PRINT VAL(B\$)	----->	RETURNS 1E-03

Notice from the above examples that VAL only operates on leading numerical data. Processing of the string evaluation terminates on the first non E character. When strings consist of alphanumeric characters with letters preceding the numerical data, the VAL function returns zero.

```
10 INPUT A$
20 X = VAL(A$)
30 IF X<1 OR X>9 THE GOTO 10
40 PRINT "YOU HAVE PRESSED A NUMBER BETWEEN 1-9"
50 GOTO 10
```

STR\$ is a very potent statement that converts a numeric expression or variable into a string. This function is invaluable in text processing and data input routines - numbers can be turned into strings, the data can then be edited and turned back into a numerical value with VAL.

One point to keep in mind, however, is when numeric data is turned into a string, a **leading blank** is inserted to allow for the sign - even though the sign is not displayed. Also, PRINT A prints the value with a **trailing blank** and PRINT STR\$(A) prints the value without a trailing blank.

```
10 A = 1650 : PRINT LEN(STR$(A)) ----> RETURNS 5
```


THE SOURCE CHAPTER THREE

```
10  A = -1650 : PRINT LEN(STR$(A))  ----> RETURNS 5
```

```
10  X = 120.62 ; Y = -120.62
20  PRINT STR$(X);LEN(STR$(X))
30  PRINT STR$(Y);LEN(STR$(Y))
40  PRINT STR$(X)+STR$(Y)
50  PRINT STR$(X+Y)
60  PRINT STR$(Y)+STR$(X)
```

Non-printable, or control characters are useful for screen formatting and for moving the cursor to another print zone without using the PRINT@ command. For instance, how many times have you seen or used Basic code that resembles the following.

```
10  PRINT@ 3,5; : INPUT "CHOOSE A NUMBER BETWEEN 1 & 10":X
20  IF X>10 OR X<1 THEN PRINT@ 3,5;" ";
30  PRINT@ 3,5;"ILLEGAL INPUT";
40  FOR I = 1 TO 100 : NEXT
50  PRINT@ 3,5;" ";
60  GOTO 10
```

The same routine can be re-written using control codes :-

```
10  PRINT@ 3,5; : INPUT "CHOOSE A NUMBER BETWEEN 1 & 10":X
20  IF X>10 OR X<1 THEN PRINT CHR$(11);CHR$(21);:ELSE GOTO 100
30  PRINT@ 3,5;"ILLEGAL INPUT"
40  FOR I = 1 TO 100 : NEXT
50  PRINT CHR$(11);CHR$(21);:GOTO 10

100 PRINT CHR$(11);CHR$(21);:PRINT@ 3,5 "O.K"
110 FOR I = 1 TO 200 : NEXT : GOTO 10
```

The above program asks for an input in the range 1 to 10. If the input is incorrect the line is cleared by using CHR\$(11) which moves the cursor up one line - a carriage return was activated when you pressed ENTER after the number - CHR\$(21) then clears to the end of the line (EOL) before printing the next message.

If you study Table 3.1 you will see that control codes support turning the cursor on/off, and it can be placed anywhere within the screen parameters simply by using the control codes.

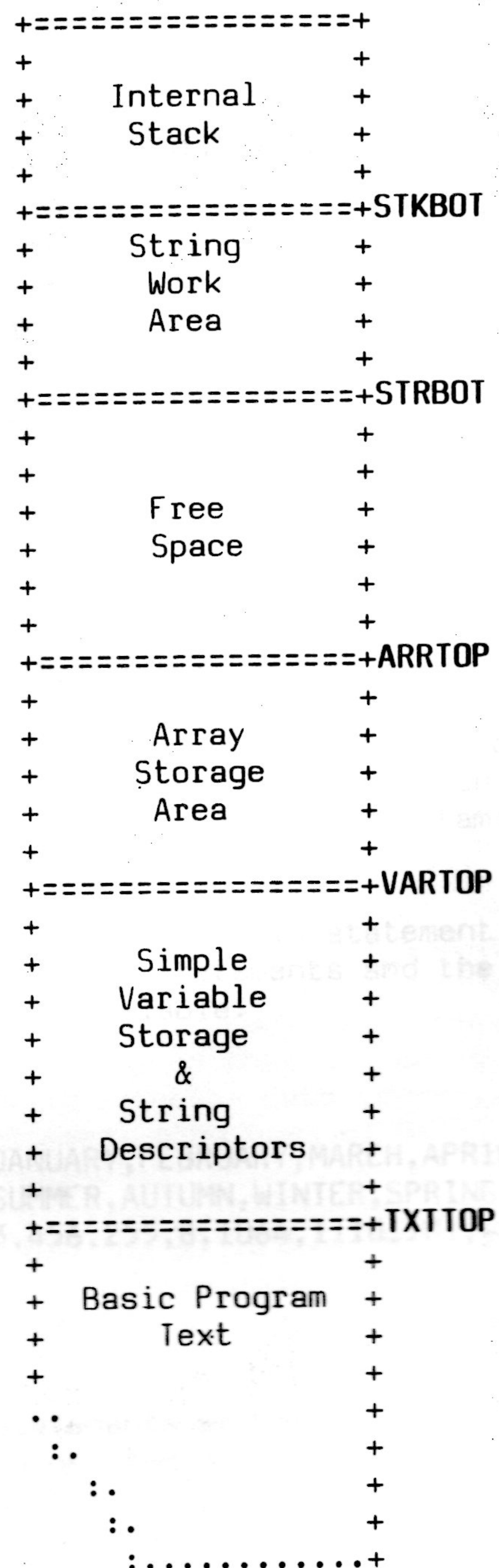
Control characters can also be embedded within strings to add some special effects to you graphic characters. This is especially helpful when you need to move a character that stretches over two vertical character positions. Try the following :-


```

10  X$ = CHR$(16)+CHR$(10)+CHR$(8)+CHR$(167)+CHR$(4)+CHR$(181)
      +CHR$(8)+CHR$(8)+CHR$(177)
20  PRINT@ 3,5;X$

```

DIA 3.2
VARIABLE & STRING STORAGE MEMORY MAP



CHAPTER FOUR

ARRAY ! ARRAY !

During the course of this chapter we shall be talking about **READ**, **DATA**, and **DIM** statements and how they are organised under Einstein Basic.

One of the most common uses of a computer is for processing data. and the simplest form of data structure available to us in Basic is the **data list** or **list of data**. Lists are something we are all familiar with - shopping lists, a list of names, a software list etc.

To create a list in Basic we use the **DATA** statement. We can include as many names or items as we wish in data statements and the only restriction is the amount of memory we have available.

```
10 DATA JANUARY,FEBRUARY,MARCH,APRIL
20 DATA SUMMER,AUTUMN,WINTER,SPRING,X
40 DATA 3,456,255,8,1064,11101101,-1,NUMBERS
```

To make use of our data statements we can use the **READ** command which tells the computer to **read one** value from a data list - **DATA & READ** are always used together. If we add the following lines to the program above we can get a good idea of how the two statements operate.

THE SOURCE CHAPTER FOUR

```

60  READ A$
70  IF A$ = "X" THEN GOTO 100
80  PRINT A$
90  GOTO 60
100 READ A
110 IF A = -1 THEN GOTO 140
120 PRINT A
130 GOTO 100
140 READ A$
150 PRINT A$

```

Line 60 tells the computer to read an item from the data list. The program checks to see if A\$ = "X" in line 70. X is put in the data list to check on how far the data pointer has moved along the list. The next items, after "X", are intended to be numeric values and this is one way of ensuring that you know what data is going into which variable. As long as A\$ is not equal to "X" the contents of A\$ are printed to the screen and the process is repeated. When A\$ = "X" program flow is directed to Line 100 where the same read procedure is repeated, but this time using numerical variable A. A similar test is performed to check when A = -1 in order to allow the next item, in our data list, which is a string value, to be read into A\$.

The computer uses a **data pointer** to keep track of item in the data list which is to be read. Every time a read is performed the pointer moves to the next item in the list.

```

10 DATA 1,2,3,4,5,6 <----.
20 FOR I = 1 TO 7          :
40 READ A                  :
50 NEXT                    :
                           :
                           :
                           :
                           :
1  2  3  4  5  6 <----+
Δ   Δ   Δ
:   :   :
I = 1 .....: I = 3   :.. I = 7
Data Pointer D.P Here Data Error
Here

```

The above program, when run, results in a DATA error message. This is because the program tried to read more items than there were in the data list.

RESTORE n will reset the pointer to the beginning of a specific data line and the command is beneficial when we need to access the data in the list more than once.

RESTORE 10

If Line 30 is changed to : FOR I = 1 TO 6 then the above statement, placed in Line 60, would reset the data pointer to the beginning of the data in

Line 10 and the next READ A would result in A = 1.

We can have any number of items in a Data List as long as they are separated by commas. As we have already discovered, data types can be mixed - string and numeric data.

More than one data item can be read from a Data list by a single READ statement, and no restrictions are placed on the number as long as there are enough items in the list. Data statements can be placed anywhere in the program and Basic will skip over them in search of the next program line.

```
10 CLS
20 DATA 1,2,3,4,5
30 DATA 9,11,13,15,17
40 READ X,Y
50 STOP
```

A point to remember when using more than one variable in a READ statement is : the data pointer still increments one place for each variable.

```
10 DATA ROBERT,0282,67890,JOHN,0434,71254
20 DATA PETE,01,24561
30 DATA GORDON,0434,68124,BOOTS,0652,33441
40 CLS
50 FOR I = 1 TO 5
60 READ N$,C,T
70 PRINT N$;" TELEPHONE ";C;"-";T
80 NEXT
```

Care must be taken to ensure that the correct variable reads the right type of data otherwise errors will be generated.

Arrays are the most powerful data structures we have available when programming in Basic. An array is fundamentally an ordered list and this list (array) can be one-dimensional, two-dimensional or multi-dimensional. Each item in the array is numbered so that it can be easily located. The number of Dimensions relates to how the data is accessed.

Arrays use **subscripted variables** which can be any of the types we have already discussed - string or numerical - but followed by a subscript which is enclosed in brackets.

```

A (9)

Δ Δ
: :.... subscript
Variable ...:
```


THE SOURCE CHAPTER FOUR

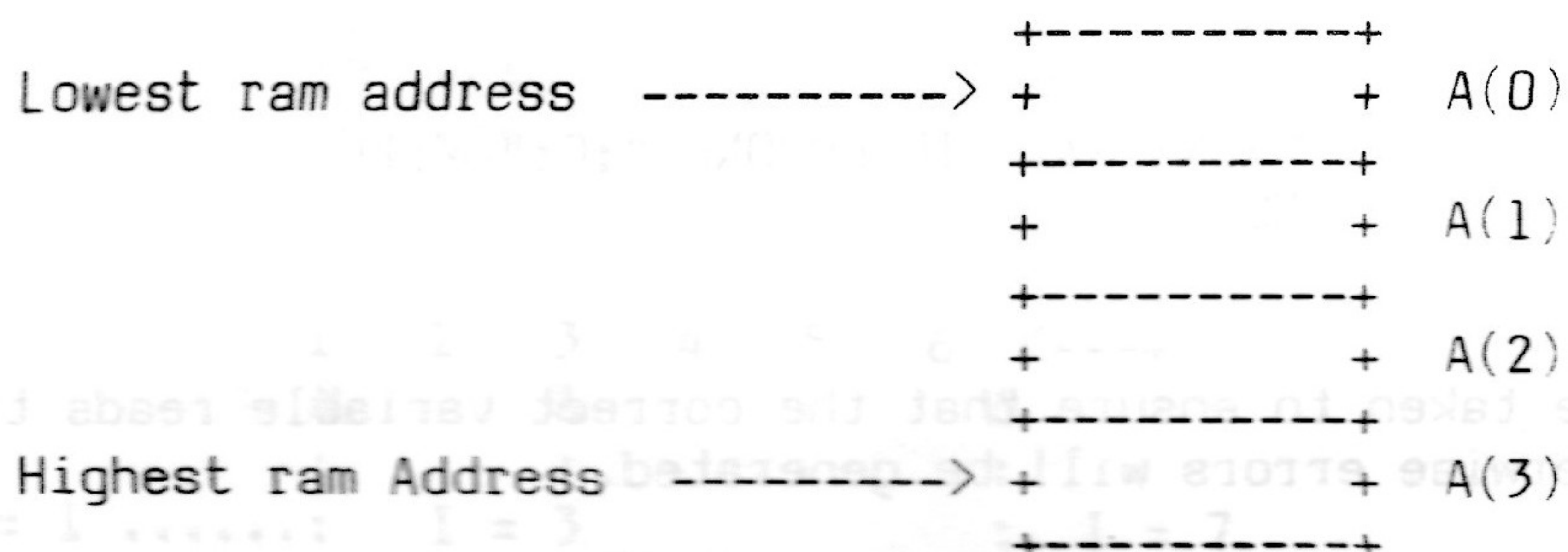
The number in brackets is the label to one of the **elements** in the array and the brackets tell the interpreter that the variable is a subscripted variable. Thus, you can store and retrieve data from one specific element in an array by telling the computer which variable and which subscript.

A(0),A(1),A(2) ... A(10) are all elements of ARRAY A

A typical example of a one-dimensional array is a software list. If you write down all the software titles you have collected, you have created a one-dimensional array. I know we said the same thing about data lists but there is a difference : a **READ** statement can only access data in a **sequential** manner and data statements cannot be modified without actually editing the Basic program. An array, on the other hand, can be accessed in a **random fashion** and data can easily be changed or modified.

The computer stores arrays, in ram, in a consecutive manner as shown in Dia 4.1. Array A is stored in memory in exactly the same order as its subscripts with A(0) being the lowest memory address and A(3) occupying the highest memory location.

DIA 4.1



```
10 CLS
20 FOR I = 0 TO 3
30   A(I) = I
40 NEXT
50 FOR I = 0 TO 3
60   PRINT "THIS IS A(";A(I);")"
70 NEXT
```

The above program will print out each element of array A with its correct subscripted value as stored in memory.

THE SOURCE CHAPTER FOUR

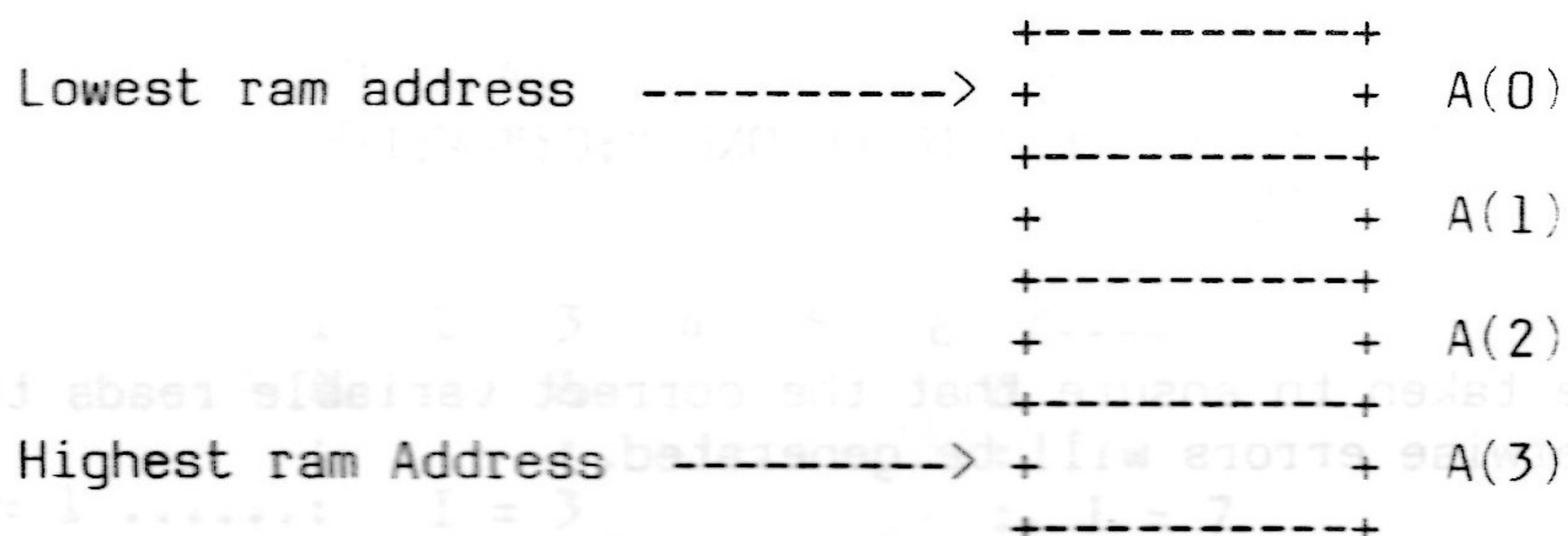
The number in brackets is the label to one of the **elements** in the array and the brackets tell the interpreter that the variable is a subscripted variable. Thus, you can store and retrieve data from one specific element in an array by telling the computer which variable and which subscript.

A(0),A(1),A(2) ... A(10) are all elements of ARRAY A

A typical example of a one-dimensional array is a software list. If you write down all the software titles you have collected, you have created a one-dimensional array. I know we said the same thing about data lists but there is a difference: a **READ** statement can only access data in a **sequential** manner and data statements cannot be modified without actually editing the Basic program. An array, on the other hand, can be accessed in a random fashion and data can easily be changed or modified.

The computer stores arrays, in ram, in a consecutive manner as shown in Dia 4.1. Array A is stored in memory in exactly the same order as its subscripts with A(0) being the lowest memory address and A(3) occupying the highest memory location.

DIA 4.1



```

10  CLS
20  FOR I = 0 TO 3
30  A(I) = I
40  NEXT
50  FOR I = 0 TO 3
60  PRINT "THIS IS A(";A(I);")"
70  NEXT

```

The above program will print out each element of array A with its correct subscripted value as stored in memory.

Another way to enter a value into an array is to use the READ & DATA statements.

```

10 CLS
20 FOR I = 0 TO 10
30 READ A(I)
40 NEXT I
50 INPUT X
60 PRINT A(X)
70 GOTO 50
80 DATA 365,400,1,-69,1.45,4000,3.333,32767,23,23.467,78

```

The above listing demonstrates how it is possible to access data stored in an array, in a random fashion. Once the data has been read into the array it is no longer necessary to use the RESTORE command. Data can be retrieved from any element as many times as is necessary, and in whatever order you choose.

In the above example that we have increased the number of subscripts to 10. This is the largest subscript we are allowed to use without taking other steps to increase the size of the array. Attempting to read data into A(11) would result in a **RANGE ERROR** being generated - this is Albert's way of telling us that we tried to access an element which does not fall within the limits of the declared dimensions. In other words, an element that didn't exist.

To create an array of more than eleven elements (subscripts 0 - 10) we must use the **DIM** statement. **DIM A(20)** tells the interpreter to reserve **twenty-one** locations (0 - 20) for storage of subscripted variable A.

Accessing data by means of an array can be likened to an index system. By using two or more arrays complex, inter-related data structures can be constructed. Let's say we wanted to create a list of all our friends and store them in the computer. We can do this, quite easily, by creating an array and writing all our friend's names into data statements.

```

10 DIM FR$(100)
20 READ B
30 FOR I = 0 TO B
40 READ FR$(I)
50 NEXT
500 DATA 12,KEITH,BARRY,TREVOR,GEOFF,ROBERT,IAN
510 DATA PAT,MIKE,SHIRLEY,KATH,COLIN,BRIAN

```

The above program provides a way of creating a list of all our friends. By placing a value at the beginning of the data we can easily add data without

THE SOURCE

CHAPTER FOUR

having to edit the main section of the program - each time we add a new friend we only need to increment the first data value. This program is fine but all it will do is print out our friend's name. The information can be extended by creating more arrays and extra data statements.

```
10 DIM FR$(100),AD$(100),TWN$(100)
20 READ B
30 FOR I = 0 TO B
40 READ FR$(I),AD$(I),TWN$(I)
50 NEXT I

500 DATA 4,GEOFF,20 CHAPEL STREET,HALIFAX,HARRY,31 TOWN PLACE,LEEDS
510 DATA BRIAN,3a THE TOWN FLATS,POOLE,NORMAN,77 WISDOM CLOSE,LONDON
```

If we wanted to find one of our friend's addresses we can simply search the name array which will give us an index into the address array.

```
60 INPUT "FRIEND'S NAME";X$
70 READ B
80 FOR I = 0 TO B
90 IF X$ =FR$(I) THEN GOTO 130
100 NEXT I
110 PRINT "NAME NOT FOUND"
120 GOTO 60
130 PRINT FR$(I);"'S ADDRESS IS ";AD$(I);TWN$(I)
140 I = B : STOP
```

Line 140 makes $I = B$ because we jumped out of a for/next loop before it had terminated. Although it isn't necessary to do this, it is good programming practice to terminate loops - sometime or other the internal stack is going to get its knickers in a twist, because, if the FOR variable, in this case I , is not cleared, the value is left on the stack.

From the above demonstration you can see that string arrays can be manipulated in the same manner as numerical arrays. The demonstration program is very simple. Complex data structures can be designed to hold more information but the principle remains the same.

ENCOUNTERS OF THE SECOND KIND

One-dimensional arrays are easy to understand because we can compare them, as we did, with something concrete like a list of names. Two-dimensional

arrays are easy to visualise - as the name implies: two-dimensional arrays can be used to represent any two-dimensional state e.g. CHESS BOARD.

The elements of a two-dimensional array are addressed by two subscripts.

One Element of a two-dimensional array $A = A(1,8)$

$\Delta \Delta$
 $: :$
 1st Subscript
 2nd subscript

A two-dimensional array can be visualised as a table made up of Rows & Columns. In general, the first subscript is the Rows and the second subscript the columns.

$A(1,8)$
 $\Delta \Delta$
 Row 1 ... Column 8

$DIM A(3,8)$ tells the computer to reserve enough space, in ram, for an array four columns by nine rows = thirty-six storage boxes.

```

10  DIM A(3,8)
20  FOR ROW = 0 TO 3
30  FOR COL = 0 TO 8
40  A(ROW,COL) = COL+ROW
50  NEXT COL
60  NEXT ROW
  
```

DIA 4.2

	0	1	2	3	4	5	6	7	8
0	+	+	+	+	+	+	+	+	+
1	+	+	+	+	+	+	+	+	+
2	+	+	+	+	+	+	+	+	+
3	+	+	+	+	+	+	+	+	+

$A(3,0)$ $A(2,4)$

THE SOURCE CHAPTER FOUR

From Dia 4.2 you can see that whenever we want to pick a particular location within a two-dimensional array we can refer to it by its Row & column number.

To READ values into the two-dimensional array we have to use nested for/next loops and we also need to set one of the loops equal to the number of rows and one equal to the number of columns.

The program listing before Dia 4.2 is an example of a nested for/next loop. In the example ROW first equals zero and COL equals zero. Location A(0,0) is filled with the value of COL+ROW. COL is incremented by one, A(0,1) is then filled with the value of ROW+COL. This process is repeated until COL = 8, here ROW and the process continues until ROW = 3. We can prove this by re-writing the above program to print out the values in each element.

```
10  DIM A(3,8)
20  FOR ROW = 0 TO 3
30  FOR COL = 0 TO 8
40  A(ROW,COL) = COL
50  NEXT COL
60  NEXT ROW
70  FOR ROW = 0 TO 3:PRINT "Row ";ROW;"=";
80  FOR COL = 0 TO 8
90  PRINT A(ROW,COL);
100 NEXT COL
110 PRINT: NEXT ROW
```

The following game listing provides us with a ready example of how a two-dimensional array can be related to a specific object. I am sure most of you will have played Connect Four or, at least, familiar with the general rules. The listing follows the original game very closely with Albert taking the role of the opponent - he plays a pretty good game too !

To play the game you are asked to choose a column between 1 & 8. If your choice is legal the computer will analyse the board before making its choice. The game continues until one player manages to place four pieces in a vertical, horizontal or diagonal row, or until there is no more space available on the playing board.

From Dia 4.3 it is obvious that the Connect Four board is very well suited to a two-dimensional array **TG\$(8,8)**. You will notice, within the program, that **TG\$(0,0)** is not used - it still exists in memory but it is our prerogative not to use it.

In the game the counters are placed on the board in a bottom to top manner - this is why the rows are numbered 8 to 1 (see dia 4.4) The row data is read into **DN(8)** array with the highest Y position first so that **DN(1)** holds

the screen cursor position 31. The X cursor position is stored in **GC(8)** and by using these two arrays screen positions can easily, and quickly, be updated.

Dia 4.3

CONNECT FOUR

8							
7							
6							
5							
4							
3							
2							
1							
	1	2	3	4	5	6	7

TG\$(5,5)

Dia 4.4

CONNECT FOUR

8							
7							
6							
5							
4							
3							
2							
1							
	1	2	3	4	5	6	7

TG\$(5,4)

R(5)=2

Do You Want Another Game?

R(3)=4

Array **TG\$(8,8)** is used to keep a copy of the screen in memory and whenever a counter is placed on the board the position is logged in the **TG\$** array. The computer looks through this array when checking if a move is legal, or deciding where to place it in memory.

Array **R(8)** is used to keep track of how many pieces are occupying positions in a particular column. The **evaluation functions** used by the computer when deciding which 'best move' to make are held in array **G(16)**. You can experiment with the evaluation constants and observe how different values affect the computer's game play.

THE SOURCE

CHAPTER FOUR

```

40 CLS
50 T$=" "
60 DIM TG$(8,8),A(4),R(8),K(4),J(4),B(16),GC(8),DN(8),X$(2),H$(2),D$(2)
70 DIM TH$(8),IN$(1):TH$=CHR$(144)+CHR$(144):TH$=TH$+TH$+TH$+TH$
80 GOSUB 570:GOSUB 290:GOSUB 470
90 GOTO 640
100 REM -----MAIN CALCULATING-----
110 D$=H$:IF X$=H$ THEN D$=C$
120 Y=1:YY=0:S=0:GOSUB 170
130 Y=1:YY=1:GOSUB 170
140 Y=0:YY=1:GOSUB 170
150 Y=-1:YY=1:GOSUB 170
160 RETURN
170 XX=1:A=1:Q=0:S=S+1
180 M=0
190 FOR I=1 TO 3:H=X+I*YY:N=R+I*Y
200 IF H<1 OR N<1 OR H>8 OR N>8 THEN GOTO 260
210 G$=TG$(N,H):IF M=0 THEN GOTO 240
220 IF G$=Q$ THEN I=3:GOTO 270
230 Q=Q+1:GOTO 260
240 IF G$=X$ THEN A=A+1:GOTO 260
250 M=1:GOTO 220
260 NEXT I
270 IF XX=0 THEN A(S)=A:K(S)=Q:RETURN
280 XX=0:YY=-YY:Y=-Y:GOTO 180
290 REM ----- DRAW THE BOARD -----
300 CLS
310 GCOL 15,0
320 BCOL 1:TCOL 15
330 J=1:FOR I= 6 TO 36 STEP 4:PRINT@I,20:PRINTJ,:PRINT@3,1+I/2:PRINTJ:J=J+1:NEXT
340 FOR I=52 TO 164 STEP 16
350 DRAW 36,I TO 228,I
360 NEXT I
370 FOR I=36 TO 228 STEP 24:DRAW I,38 TO I,164:NEXT I:DRAW 36,38 TO 228,38
380 PRINT@9,1;"C O N N E C T - F O U R"
390 RETURN
400 REM ----- MAIN INPUT ROUTINE -----
410 PRINT@5,22:TCOL 0:PRINTCHR$(5)
420 BCOL 1:TCOL 15:PRINT@6,22;MES$;
430 IN$=INCH$
440 IF IN$="" THEN GOTO 430
450 BEEP
460 RETURN
470 REM ----- DEFINE CHARACTERS -----
480 SHAPE 140,"1C 3C 64 FC DC 60 3C 1C"
490 SHAPE 141,"E0 F0 C0 FC EC 10 F0 E0"
500 SHAPE 142,"3C 74 F4 FC FC E0 7C 3C"
510 SHAPE 143,"F0 B0 BC FC FC 1C F0 F0"
520 SHAPE 144,"FF FF FF FF FF FF FF FF"
530 H$=CHR$(140)+CHR$(141)
540 C$=CHR$(142)+CHR$(143)
550 RETURN
560 REM ----- INITIALISE VARIABLES -----
570 DATA 7,18,11,16,15,14,19,12,23,10,27,8,31,6,35,4
580 FOR I=1 TO 8:READ GC(I),DN(I):NEXT I

```


THE SOURCE
CHAPTER FOUR

590 EM -VALUES BELOW CONTROL COMPUTER
600 EM -TRY CHANGING THEM AND SEE HOW
610 ATA 1,120,505,1E22,1,880,3000,1E30,1,80,1000,1E16,1,475,3050,1E14
620 ORI=1T016:READ B(I):NEXTI
630 ETURN

-EVALUATIONS-

-IF EFFECTS THE PLAY.-

```

640 BEEP:BEEP:ME$="DO YOU WANT TO GO FIRST?":GOSUB 400
650 IFIN$="Y"THENGOTO680
660 IFIN$<>"N"THENGOTO640 ELSE I=INT(RND(8)+1):GOTO1140
670 REM ---- HUMAN ROUTINE ----
680 BEEP:FORI=1T0500:NEXT:ME$="PICK A NUMBER (1 TO 8):- "
690 GOSUB 400
700 X=INT(VAL(IN$))
710 IFX>1 AND X<=8 THEN GOTO 740
720 PRINT@5,22:PRINT"                ":PRINT@5,22
730 PRINT"ILLEGAL INPUT,PLEASE TYPE 1-8":VOICE 1,0,15,30,5,0:MUSIC "V1 DDF":VOICE 1,0,0,5,5,5:PRINT@20,22;"
740 R=R(X):IF R>7 THEN GOTO 720
750 R(X)=R+1:R=R+1:E=GC(X):F=DN(R):TG$(R,X)=H$
760 PRINT@E,F:TCOL 3:PRINTH$;:TCOL 15
770 X$=H$:BEEP:GOSUB 100
780 FORI=1T04:IF A(I)<4 THEN GOTO 820
790 I=4
800 FORI=1T06:PRINT@5,22:PRINT"<<< O.K YOU WIN !!! >>>";:FORA=1T01000:NEXTA
810 PRINT@5,22;"                ";:FORA=1T01000:NEXTA:NEXTI:GOTO1280
820 NEXTI
830 REM ---- COMPUTERS ROUTINE ----
840 P6=0:ME$="THINKING ":PRINT@5,22;"
850 TCOL15:PRINT@5,22:PRINTME$
860 Z1=13:Z2=22:TCOL15:PRINT@Z1,Z2;TH$;
870 U=0:J=1:FORP=1T08:R=R(P)+1
880 IF R>8 THEN GOTO 1090
890 E=1:X$=C$:F=0:X=P
900 GOSUB 100
910 FORL=1T04:J(L)=0:NEXT
920 FORI=1T04:A=A(I):IF A-F>3 THEN I=4:GOTO1140
930 B=A+K(I):IFB<4 THEN GOTO 950
940 E=E+4:J(A)=J(A)+1
950 NEXTI
960 FORI=1T04:W=J(I)-1:IF W=-1 THEN GOTO990
970 Z=8*F+4*SGN(W)+I
980 E=E+G(Z)+W*6(8*F+I)
990 NEXTI
1000 IFF=1 THEN GOTO 1020
1010 F=1:X$=H$:GOTO900
1020 R=R+1:IFR>8THENGOTO1050
1030 GOSUB 100
1040 FORI=1T04:IF A(I)>3THENE=2: ELSE NEXTI
1050 IFE<U THEN GOTO 1090
1060 IFE>U THEN D=1: GOTO 1080
1070 D=D+1: IF RND (8) >1/0 THEN GOTO 1090
1080 U=E:P6=P
1090 GCOL 8,6:PRINT@Z1,Z2;T$;:Z1=Z1+1:BEEP:NEXTP

```


THE SOURCE
CHAPTER FOUR

```
1100 IF P6<>0 THEN GOTO 1130 ELSE PRINT@10,22:TCOL 0:PRINT"  
1110 PRINT@5,22:TCOL 5:PRINT"~~~ IT'S A DRAW ~~~";FORL=1TO2000:NEXTL  
1120 GOTO1280  
1130 X=P6  
1140 PRINT@5,22:TCOL 0:PRINT"  
1150 PRINT" I'M GOING IN COLUMN";X;  
1160 R=R(X)+1:R(X)=R(X)+1  
1170 T6$(R,X)=C$  
1180 X=C$  
1190 E=6C(X):F=DN(R):BEEP  
1200 FORI=1TO3:PRINT@E,F:TCOL 8:PRINT" ";:FORL=1TO400:NEXT:PRINT@E,F  
1210 PRINTC$;:FORL=1TO400:NEXT:NEXT:BEEP:FORT=1TO100:NEXT  
1220 GOSUB 100  
1230 FORI=1TO4:IF A(I)<4 THEN NEXTI: GOTO 670  
  
1240 I=4  
1250 PRINT@5,22:PRINT"  
1260 FORI=1TO8:PRINT@10,22:TCOL 7:PRINT"---- SORRY I WIN ----";:FORJ=1TO300:NEXT  
1270 PRINT@10,22:TCOL 13:PRINT"!!!! HA! HA! HA! ////";:FORJ=1TO300:NEXT:NEXT  
1280 FORI=1TO2000:NEXT  
1290 MES$="DO YOU WANT ANOTHER GO?":GOSUB 400  
1300 IFIN$="Y"THENCLEAR:GOTO40  
1310 IFIN$<>"N"THENGOTO1290  
1320 CLS:STOP
```

Ready

As you can imagine, there are hundreds of items stocked by a worthwhile ironmonger but Steve's biggest problem was keeping track of how many screws he had in stock. He kept a range of screws from 1/16th of an inch up to 8 inches in 1/16th of an inch increments. Steve is a tidy sort of chap and stores his screws in draws similar to Dia 5.1

DIA 5.1

```

+---+---+---+---+
1/16th screws -> :   :   :   :
+---+---+---+---+
:       :       :       :       :
+---+---+---+---+
:       :       :       :       :
+---+---+---+---+
:       :       :       :       : <-- 1" screws
+---+---+---+---+

```


THE SOURCE

CHAPTER FIVE

Stephen is a logical thinking fellow and it wasn't long after he purchased a shiny, new Einstein that he realised it was possible to write a program that would keep a record of all his screws simply by using an array to hold the data.

His screws were kept in two cabinets. Each cabinet had four draws and every draw stored screws, of every size, up to one inch increments so that drawer one held screws from 1/16th of an inch to 1 inch. Draw two held screws from 1 & 1/16th of an inch to two inches, and so on up to draw eight which held screws from 7 & 1/16th of an inch to eight inches.

Steve had never used multi-dimensional arrays but this is how he thought out his preliminary program.

```

          SCW(4, 4, 8, 2)
              Δ Δ Δ Δ
              : : : :.....cabinet
row .....: : :
column .: :
              :..... draw

```

```

10  DIM SCW(4,4,8,2)
20  FOR CAB = 1 TO 2
30    FOR DRAW = 1 TO 8
40      FOR ROW = 1 TO 4
50        FOR COL = 1 TO 4
60          READ STOCK
70          SCW(ROW,COL,DRAW,CAB) = STOCK
80        NEXT COL
90      NEXT ROW
100    NEXT DRAW
110  NEXT CAB
120  REM PUT REST OF PROGRAM HERE WHEN I GET IT WORKING

500  DATA 3600,2000,4000,12000,1100,4536,2139,1921
510  DATA 1200,1324,5236,543,123,4312,1568,612
520  DATA 129, 1345, and so on .....

```

Once he had filled the array with the stock values Steve knew that he could find the amount of stock for any particular screw by accessing the correct element of the array.

Element SCW(1,1,1,1) held the 1/16th screw

Element SCW(1,1,2,1) held the 1 & 1/16th screw

Element SCW(1,1,4,2) held the 7 & 1/16th screw

By adding 'bells and whistles' he eventually ended up with a neat program that allowed him to check his stock at any given time. Of course, with the Einstein, there are far better ways of writing this program - what about the disc ? However, it serves to show how a multi-dimensional array should be organised, and how those frightful for/next loops should be nested.

PART TWO

```
10 CLS
20 FOR I = 128 TO 255
30 INPUT "Press Any Key For Next Cursor: "
40 POKE AFBX,I
50 NEXT I
```

Be
to
con

commands with a
range between 128 and 255
be sure to save

CHAPTER SIX

POKEING AROUND

It's now time for us to take a look inside the computer. No ! Put the screw-drivers away ! I am speaking metaphorically. We shall look inside the computer with the aid of the Einsteins's Basic commands : **PEEK**, **POKE** and **VPEEK**, **VPOKE**.

With the assistance of **PEEK**s & **POKE**s we can access memory locations directly and these statements can be used to great advantage within our programs.

Try the following little program and you will see how **Poke** can be used to directly modify memory.

```
10 CLS
20 FOR I = 128 TO 255
30 INPUT "Press Any Key For Next Cursor ";X
40 POKE &FB3F,I
50 NEXT
```

Before we can use these two commands with any significance it is necessary to understand the difference between ROM and Ram. Also, to utilise the commands to our advantage, we must be aware of what happens within the computer.

THE SOURCE CHAPTER SIX

ROM AND RAM

ROM or Read Only Memory is that part of memory which can only be read -you cannot alter the contents of ROM and poking has no effect. Ram or Random Access Memory, on the other hand, can be read or overwritten - you can peek or poke ram. Random means that we can access memory locations directly as opposed to sequential which only allows us to start at the beginning and read or write each location in turn.

When looking at memory the situation is further complicated by the fact that it is impossible to tell whether we are looking at ram or ROM, and it becomes essential to understand which part of memory is ram and which is ROM. We must also be careful of what, and where we poke - we have already seen the effect a poke can have on the operating system when we tried the short program listed above. Some blocks of memory are used by the system and if pokes are used in a careless manner it is very easy to cause a "system crash" or "lock-up". Although we cannot harm the computer with our poking, if the system does crash our only recourse is to switch off and start again. This process is not at all satisfactory, particularly if we have just finished typing in a long program. So our first golden rule is : whenever we use pokes save the program to disc before attempting to run it. Then, if the computer does crash, we have, at least, the comfort of knowing we don't have to re-type the whole program.

Organisation of memory is complicated by the fact that the computer has **64K of Ram** and **16K of ROM** with the facility to add another 16K of ROM. The Z80 Cpu is only capable of addressing 64K of memory at any one time which means that some method of switching memory must be employed so that ram and ROM are always available to the user. Albert gets around this problem by using OUT (24H) which switches out the bottom part of ram and then switches in Mos. This is how we are able to use Mcals [Machine Calls] from within our machine code programs - more about this later.

The Mos is located from 0000 - 288FH in the lower part of memory and is responsible for handling such processes as Vdp, disc controller, printer etc.

Dos [Disc Operating System] is loaded from the system tracks of the disc and this program, when installed, resides in ram (mos is on ROM) from location zero. There are various tables in this section of memory that inform the computer where to find the relevant routines to carry out such operations as read/write to disc, print to the display, load programs and many more activities.

Basic, on the Einstein, is loaded into memory in exactly the same way as we would load a game or utility. Once loaded, into ram, Basic takes over the management of the computer and by "talking" to Mos and Dos it does all those wonderful things you ask it to do like file handling, moving sprites, drawing lines, to name but a few.

DIA 6.1
MEMORY ORGANISATION [BASIC]

```

+=====+
:   Scratchpad Area   :
:   Starts at FBOOH   :
+=====+ .. Topram PTR 21
:   Free Space For   :
:   Machine Code Routines :
+=====+ .. Limit  PTR 20
:   Used for internal Vdu :
:   Accessed by the EDITOR :
+=====+ .. Vram  PTR 19
:   BASIC STACK      :
:   Moves DOWN in memory :
+=====+ .. Stkbot PTR 18
:   String work area :
:   Stores actual String :
:   pointed to by STRING :
:   DESCRIPTOR BLOCK :
:   Builds DOWN in memory :
+=====+ .. Strbot PTR 17
:   FREE SPACE      :
:                   :
:   The Stack & String :
:   Work Area ALL BUILD DOWN :
:   into this area ... :
:                   :
:   Program Text : Variables :
:   Arrays.      ALL BUILD UP :
:   into this area .. :
:                   :
+=====+ .. Arrrtop PTR 16
:                   :
:   Array Area      :
:   Builds UP in memory :
+=====+ .. Vartop PTR 15
:   Simple Variables :
:and String Descriptor block:
:   Builds UP in memory :
:   each value is ended with &7F :
+=====+ .. Txttop PTR 14
:   Program Statement :
:   Table              :
Start of Ebasic :   Build UP in memory :
:4001H..+=====+ .. Htext  PTR 0
Start of Basic..3E01H..+=====+
:   BASIC INTERPRETER :
+=====+
:   OUT (24H) MOS SWITCH :
+=====+ .. 0038 Hex
:   DOS routines       :
+=====+ .. 0000 Hex

```


Because Basic is loaded into ram memory it becomes a very powerful programming tool. Think about it for a moment. Einstein Basic can be altered in any way we see fit (within reason) and the new version can be re-saved to disc and used as a customised version - we can add as many new commands as we wish within the practicalities of memory. Before we do this, however, we need to understand how the Basic interpreter operates.

SYSTEM VARIABLES

Roms can never change their memory contents and in an ever changing environment, such as Basic, Mos and Basic must have a method of storing various pieces of information : length of the Basic program, background colour, cursor position, which drive is in use, etc. etc.

The Einstein keeps track of this information by reserving a part of ram at the top of memory. This is known as the **Scratchpad Area** and starts at FB00H. Basic also has its own scratchpad which it stores in the bottom of memory from 0106H through to 0281H. These two areas of memory are extremely critical for the correct operation of the Einstein, and extreme caution should be exercised when altering these regions of memory. We have already experimented with changing the cursor character simply by modifying a value in the scratchpad area. Now, try this small program : -

```
10 CLS
20 FOR I = 2 TO 15
30 POKE &FB38, I*16+1
40 PRINT "COLOUR ";I
50 NEXT I
```

Line 30 follows the formula : **Text Colour * 16 + Background Text Colour**
To aid understanding try changing Line 30 to read :-

```
POKE &FB38, I*16+(I-2)
```

Now re-run the program. Try variations of your own until you are sure, in your own mind, what is happening.

THE PROGRAM STATEMENT TABLE

The Program Statement Table contains the source statements of the actual Basic lines that have been typed in from the keyboard or loaded from disc. The start address of this area is fixed by Basic at 4001H (256) and 3E01H (Einstein) but its ending address will vary with the size of the program currently in memory. Whenever a program line is added or deleted the end of the Pst is adjusted accordingly.

As each line of a Basic program is entered from the keyboard it is scanned for reserved words and if any are found they are tokenised and stored in a compressed format. Tokens are a way of saving memory - it is far more economical to store the token B6H than the Basic command UNLOCK because this method saves five bytes of memory. A basic program has many lines, most lines have multiple statements so the saving on memory is vast - as much as 20%. If you are curious and want to see the tokens yourself type in the following program : -

```
10 CLS
20 FOR I = &3A0D TO &3C33
30 X = PEEK(I)
40 IF X> &7F THEN PRINT " ";
50 PRINT CHR$(X AND $7F);
60 NEXT I
```

**** 256 OWNERS USE &3BC7 TO &3E0E**

Line 20 equates I to the first location of the token list. Variable X then peeks this address and returns a number which is tested for a value greater than &7F the Basic interpreter detects when it has come to the end of a reserved word by testing bit seven of the letter which is set (1) if it is the start of a new reserved word.

```
A = 41H = 0100 0001
set bit 7 = C1H = 1100 0001
```

SPC(STEP

Stored in memory as --> 0D3,50,43,28,0D3,54,45,50
 S P C (S T E P
 Δ Δ
 :.Bit 7 Set .:

If X is greater than &7F then a space is printed because we know it is the start of a new token. Each value is anded with &7F in order to print out the correct Ascii character. Delete line 40 and re-run the program. See the difference ?

As soon as the program enters the RUN mode control is passed over to the Execution Driver which scans each statement for tokens and, if one is found, the Execution Driver passes control to a routine that deals with the command or function.

When the computer spots a token e.g. GOTO, it knows immediately which routine and the actual location of the routine in memory. The secret of this lies in the way tokens are allocated to the various Basic statements.

THE SOURCE
CHAPTER SIX

KEYWORD	TOKEN	KEYWORD	TOKEN
SPC	&6F	STEP	&70
TAB	&71	TO	&72
THEN	&73	+	&74
-	&75	Δ	&76
*	&77	/	&78
MOD	&79	AND	&7A
OR	&7B	XOR	&7C
>	&7D	=	&7E
<	&7F	AUTO	&80
CHAIN	&81	CLEAR	&82
CLOSE	&83	CLS	&84
CONT	&85	CREATE	&86
DATA	&87	DEF	&88
DEL	&89	DIM	&8A
DOKE	&8B	DRIVE	&8C
ELSE	&8D	END	&8E
FOR	&8F	GOSUB	&90
GOTO	&91	HOLD	&92
IF	&93	INPUT	&94
LET	&95	LIST	&96
LOAD	&97	MGE	&98
MOS	&99	NEW	&9A
NEXT	&9B	OFF	&9C
ON	&9D	OPEN	&9E
OUT	&9F	POKE	&A0
POP	&A1	PRINT	&A2
READ	&A3	REM	&A4
RENUM	&A5	UNPLOT	&A6
RESTORE	&A7	RETURN	&A8
RUN	&A9	SAVE	&AA
PLOT	&AB	STOP	&AC
SWAP	&AD	VERIFY	&AE
WAIT	&AF	FMT	&B0
APPEND	&B1	DIR	&B2
ERA	&B3	LOCK	&B4
REN	&B5	UNLOCK	&B6
MUSIC	&B7	CALL	&B8
IOM	&B9	NULL	&BA
PTR	&BB	SEP	&BC
SPEED	&BD	WIDTH	&BE
ZONE	&BF	TI\$	&C0
TEMPO	&C1	VOICE	&C2
PSG	&C3	ABS	&C4
ASC	&C5	ATN	&C6
CHR\$	&C7	COS	&C8
DEEK	&C9	EVAL	&CA
EXP	&CB	HEX\$	&CC
INP	&CD	INT	&CE
LEN	&CF	LN	&D0
LOG	&D1	PEEK	&D2
POINT	&D3	POS	&D4

KEYWORD	TOKEN	KEYWORD	TOKEN
RND	&D5	SCRN\$	&D6
SGN	&D7	SIN	&D8
SQR	&D9	STR\$	&DA
TAN	&DB	VAL	&DC
LEFT\$	&DD	MID\$	&DE
RIGHT\$	&DF	ERR	&EO
ERL	&E1	EOF	&E2
FN	&E3	INCH	&E4
KBD	&E5	MUL\$	&E6
NOT	&E7	PI	&E8
SIZE	&E9	VSTAT	&EA
DRAW	&FF80	TCOL	&FF81
GCOL	&FF82	BCOL	&FF83
SPRITE	&FF84	MAG	&FF85
SHAPE	&FF86	ORIGIN	&FF87
ELLIPSE	&FF88	DOS	&FF89
RAD(&FF8A	DEG(&FF8B
POLY	&FF8C	FILL	&FF8D
VPOKE	&FF8E	VPEEK	&FF8F
VDOKE	&FF90	VDEEK	&FF91
BEEP	&FF92	BIN\$	&FF93
RST	&FF94	KEY	&FF95
ADC(&FF96	BTN(&FF97
PSW	&FF98	SCREEN *	&FF99
CHAR *	&FF9A	PCOL *	&FF9B
VDP *	&FF9C	JOY(*	&FF9D
BAUD *	&FF9E	MODE *	&FF9F

* Denotes 256 command

The Basic interpreter contains a Verb Action List which holds the addresses for all the routines used by the basic commands. To find the correct address the computer has to calculate the displacement into this list according to the value of the token. Basic has two Verb Action Lists. One is the **User Command Table** and all tokens referring to this list are prefixed with **&FF** (the **&** is another way of saying this is a hexadecimal number) so that the Execution Driver knows in which table to look. If the token is not prefixed with **&FF** a further test is carried out to find the correct displacement into the main Verb Action List.

The very first command token in the standard Command Table (Verb Action List) is **AUTO (&80)** and all other tokens are increments of this number. For example, **PRINT** has a token value of **&A2** and to find the address of the **PRINT** routine, the computer subtracts **&80** from the token, multiplies the result by two because each address is two bytes in length, and then adds this value to the start of the Verb Action List. The Execution Driver now extracts the address from this location and jumps to the routine that will carry out the print command.

THE SOURCE

CHAPTER SIX

Immediately following the Program Statement Table is the Variable List Table (VLT). This table contains the names and values of all variables currently initialised within the Basic program. The list is divided into two sections: simple variable names including string descriptor blocks and subscripted variables. This table is dynamic and will alter every time a variable is declared or deleted.

STRUCTURE OF A BASIC LINE

```

06 : 00 : 0A : 00 : Actual Basic Statement & Tokens : 00 : 0E :
-Δ-+Δ-+Δ-+Δ-+-----+Δ-+Δ-+
:....:      :....:                                     :
:                                     :
Offset      :... Line Number      End of Line Marker ...:
to next     :                   Start of next line .....:
Basic line

```

Basic statements are decoded in a simple manner : -

```
20 PRINT "TEST"
```

The above statement would be store in memory as : -

[illegible]

We can use Pokes to put our machine code routines into memory. The Poke statement can also be used to pass parameters to a machine code routine. But before we proceed, let's take a look at how to set up a Poke or Peek routine from Basic. We will then examine two important rules that must be used when using these commands.

Example of Pokeing to sequential addresses.

```

10  X = START_ADDRESS
20  Y = END_ADDRESS
30  FOR I = X TO Y
40  READ A
50  POKE I,A
60  NEXT I
70  STOP
80  DATA &34,&45,&76,&82,&0D, etc.

```

Routine to Poke data to different addresses.

```

10  READ ADR, CODE
20  IF ADR = -1 AND CODE = -1 THEN END
30  POKE ADR, CODE
40  GOTO 10
50  DATA ADR1,&00, ADR2,&0CD, ADR3,&78, ADR4,&0ED, -1, -1

```

Where ADR1....ADR4, in the above example represent memory locations. A check for -1 is there to endure that the end of data has been reached. The reason CODE is also checked for -1 one is simply that it is possible for an address to be -1 as we shall now discover.

Two Rules for PEEK & POKE.

In Chapter Two we unearthed the method of how the Einstein stores 2-byte numbers : Least significant byte/Most significant byte. Whenever we read a 16-bit (2-byte) value from memory we have to correct for this anomaly by using the formula listed below. O.k. I know you can use DEEK and DOKE but not all Basics are as complete as our Einstein Basic, so it is wise to be in the know.

RULE 1

$$\text{VALUE} = \text{MSB} * 256 + \text{LSB}$$

or

$$\text{PEEK(AD)} + \text{PEEK(AD+1)} * 256 = \text{VALUE}$$

If we are using decimal numbers to represent our memory addresses, we must use the following rule for all addresses over 32767.

THE SOURCE CHAPTER SIX

POKE ADDRESS = (DECIMAL ADDRESS - 65536)

The reason for the above rule is related to unsigned 16-bit integers and the two's complement of a number but you know all about this from our diatribe in Chapter Two ... Don't you?

Decimal addresses range from 0 - 65535. However, signed integers, as used, internally, by the computer, may only be within the range -32768 to +32767. Now, here's the anomaly, the signed integer -1 = &FFFF and the unsigned integer &FFFF = 65535 - this is where the confusion starts! Take a look at the following list :-

DECIMAL	HEXADECIMAL	POKE VALUE
32767	&7FFF	32767
32768	&8000	- 32768
32769	&8001	- 32767
65535	&FFFF	-1
65534	&FFFE	-2

Now is a good time to try out a few Peeks and discover some of the Einstein's secrets. The program, listed below, will allow you to Peek anywhere in memory. It will display the memory location followed by whatever is in that memory byte. If the value in memory has an equivalent Ascii value it will be printed otherwise, the program will print the actual value represented in memory. This program is not a disassembler but it will allow you to investigate a few interesting areas of Basic.

```

10  REM MEMORY PEEKER
20  INPUT "START ADDRESS ";SA
30  INPUT "END ADDRESS ";EA
40  CLS : FL = 0
50  FOR L = SA TO EA
60  PK = PEEK(L)
70  GOSUB 200
80  NEXT
90  PRINT : INPUT "ANOTHER RUN ? ";A$
100 IF A$ = "Y" THEN 40 : ELSE 20
199 REM SORT OUT ASCII FROM CODE
200 IF PK<33 THEN 250
210 IF PK>122 THEN 250
220 IF FL = 0 THEN PRINT L;" ";PK;" ";CHR$(PK); : GOTO 240
230 PRINT CHR$(PK);
240 FL = 1 : RETURN
250 GOSUB 300 : PRINT L;" ";PK;" ";-
260 FL = 0
270 RETURN
300 IF FL = 1 THEN PRINT : RETURN
310 RETURN

```


Try looking at memory areas around &3A06 (&3BBE for 256 owners). Did you recognise the reserved word list? Remember that the first word has bit-7 set, this is why it is not printed by the program. Now take a look at &3E01 (&4001 256 users) and you should see all the data of the above program.

Problem.

Poke memory address 32769 with the value 128

```
10 POKE 32769 - 65536,128 ..... Method already discussed
```

```
10 POKE &8001,128 .... Using hexadecimal for address
```

Using the latter method simplifies the problems considerably but it is wise to understand all method - this is very true if you are trying to convert a program from another machine - remember not all machines have the luxury of Einstein's Basic.

DOKE

The DOKE command allows us to poke into memory without having to use Lsb/Msb methods. For example :-

```
10 DOKE &3FF0,12987
```

DOKE will automatically store the number in the correct format, and in the correct address.

DEEK

DEEK works in exactly the way as DOKE. It will automatically reverse the numbers, extracted from memory, and return the correct decimal number. If memory address &3FF0 held the value 209 and address &3FF1 contained 135 the following program would result in variable X = 34769.

```
10 CLS
20 X = DEEK(&3FF0)
30 PRINT X
```

VPOKE, VPEEK, VDOKE, VDEEK are all related to the video display and to use these commands effectively we need to understand how the video chip stores its data. The video chip is explained in great detail in a subsequent chapter.

CHAPTER SEVEN

By now, I am sure, that we all agree that Basic is very versatile - its is reasonably easy to use, has an adequate instruction set, and is dynamic by allowing 'user' functions to be added to the internal structure of the Interpreter. Unfortunately, it is not easy to write routines, that add new commands, without a working knowledge of assembly language, and how the Interpreter operates. We have already investigated some of the inner secrets of the Interpreter and so, at this point, it would be prudent to examine some of the more elementary concepts of assembly language programming.

The following is not an attempt to 'teach' you how to program in this low-level language, it is included in order that you can, at least, follow the subsequent explanations on customising Basic. If you have a desire to learn assembly language, I suggest that you purchase a 'sister' book, to this, titled "A Foundation Course in Programming the Z80" which deals in detail with this subject.

Basic is a 'nice' programming language, and it is an ideal medium for learning the art of programming computers - although my personal preference would be 'C'. However, Basic, by machine code standards, is slow. Each command, as we have already learned, must be analysed by the Interpreter before being executed. There are also many tasks that cannot be successfully accomplished from Basic but are easily implemented in machine code. Anyway, down to business.

A BRIEF OVERVIEW OF THE Z80

The Z80 contains two sets of eight-bit internal, general purpose

THE SOURCE CHAPTER SEVEN

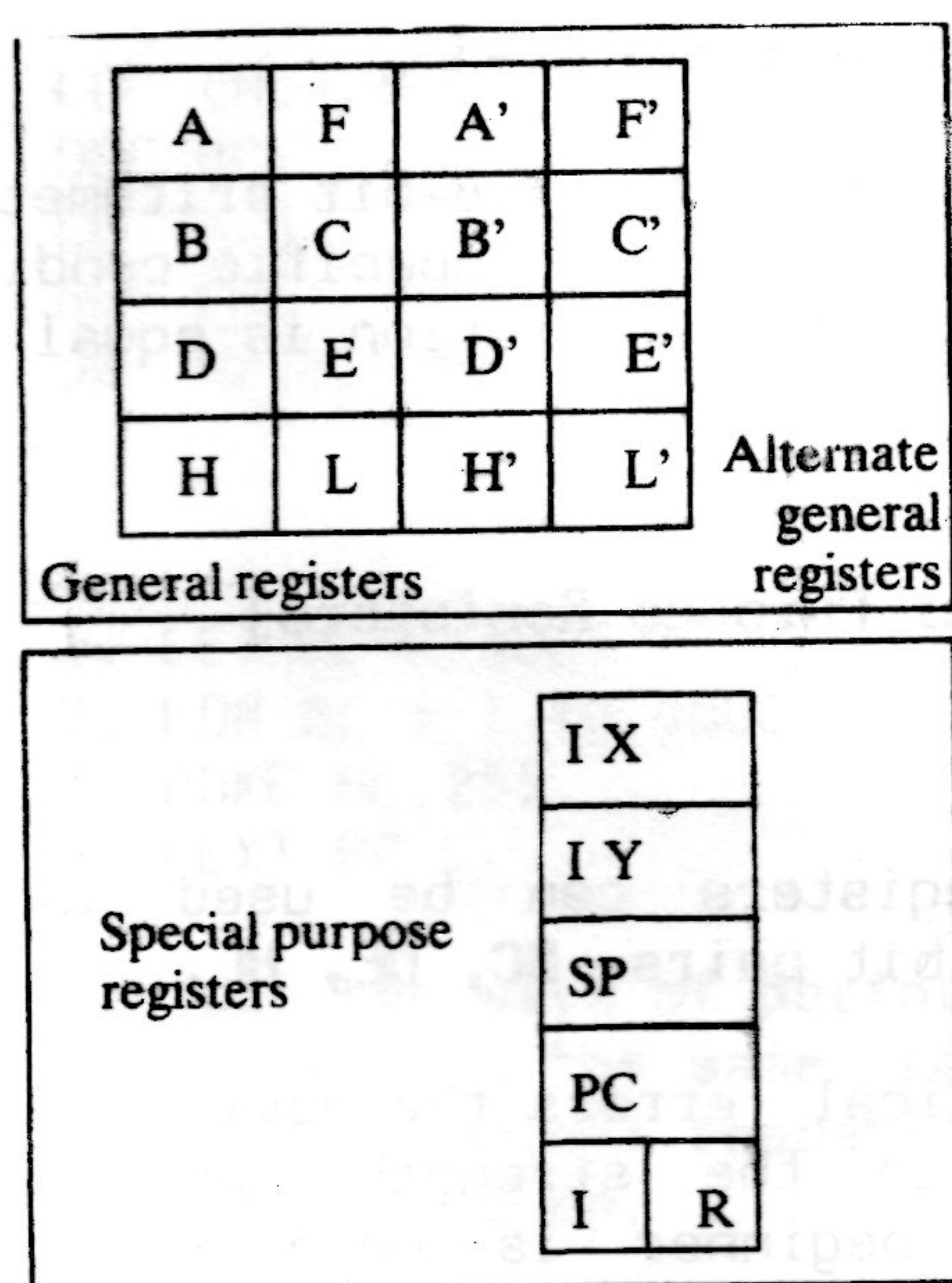
registers:-

A B C D E F H L

The second set are called alternate registers, and are designated the sign ' after the register, to signify that it is an alternate register. The alternate registers are named exactly as the main set : -

A' B' C' D' E' F' H' L'

This alternate set can only be accessed by the two instructions EX AF,AF' & EXX. When executed they exchange the contents of the main set with those of the alternate set of registers. Only one set of registers may be used at any one time.



Apart from the general purpose registers, which are eight-bit registers, there are four **sixteen-bit** registers IX, IY, PC, SP. The I & R registers are special registers.

PC [Program Counter]

The program counter is used , internally, to keep a record of which instruction is to be executed after the current instruction has finished.

SP [Stack Pointer]

The stack pointer holds the sixteen-bit address of the current top of stack. **The Stack** can be located anywhere within ram and is organised as a **last in, first out** file. Data can be pushed onto the stack or popped off the stack but the data can only be accessed in a sequential manner, and the first byte popped from the stack will be the very last byte pushed onto the stack.

IX & IY [Index Registers]

The two, independent, index registers hold a sixteen-bit base address that is used in index addressing modes. In this mode, an index register is used as a base point to a region of memory from which data is to be stored or retrieved. A displacement byte is then included to specify the memory location. LD A,(IX+02).

A & F [Accumulator and Flag Registers]

The accumulator holds the results of 8-bit arithmetic or logical operations while the flag register indicates specific conditions such as indicating whether or not the result of an operation is equal to zero.

B, C, D, E, H, L [General Purpose Registers]

These general purpose registers can be used individually as eight-bit registers or as sixteen bit pairs, BC, DE, HL.

One of the most critical errors the novice usually makes is due to the lack of familiarity with the strengths and weaknesses of the Z80's instruction set. The beginner is more likely to select unsuitable instructions and use less desirable registers than the experienced programmer - practice and familiarity are the only ways to overcome this phase of the learning cycle.

Selecting the correct register is often confusing to the beginner, so it is very important to understand the limitations of each group of instructions.

A programming problem may be solved by writing routines in many different ways. Two programmers working severally on the same problem will more than likely solve the problem by different programming methods.

THE SOURCE CHAPTER SEVEN

Example:

FILL MEMORY FROM &4007 THROUGH TO &8007 WITH &FF

```
LD HL,&4007      ; 1ST MEMORY LOCATION
LD DE,&4008      ; DESTINATION MEMORY
LD BC,&4000      ; COUNTER
LD (HL),&FF     ; FILL MEMORY LOCATION WITH &FF
LDIR            ; BLOCK MOVE, IMPLEMENT AND
               ; REPEAT
```

Study the above example until you are satisfied that you know how this routine works.

The same routine could be written as follows:

```
BLOOP:
LD HL,&4007      ; 1ST MEMORY LOCATION
LD BC,&4000      ; COUNTER
LD (HL),&FF     ; FILL MEMORY WITH &FF
DEC BC
LD A,B          ; GET HIGH OF COUNT INTO A
OR C            ; CHECK BC FOR ZERO
JR NZ,BLOOP     ; NO THEN DO IT ALL AGAIN
```

Basic:

1. LET HL = 4007
2. FOR BC = 1 TO 4000
3. POKE HL,255
5. NEXT BC

The latter routines are just two ways of performing the same task. There are MANY other ways of accomplishing the same result. Which routine to use would depend on many variables: readability, program memory space, assembler memory space, speed and so on.

The most important thing, as far as you are concerned is to get the routine written and not to worry about speed, logical operations and unfamiliar instructions that could have helped to produce faster and tighter code. Some programming methods are more desirable than others, but never waste time searching for the best way - there is always a better way! By all means try to tighten code which you feel is slow and unwieldy but never strive for the perfect solution - there isn't one.

Learning Z80 is made slightly more difficult than other processor codes due to the large instruction set available but this large instruction set also allows powerful operations not available on other CPU's.

Because the size and execution speed of a program is more likely to be directly affected by the selection of registers it will serve us well, at this stage, to spend a little time examining their various functions.

Z80 REGISTERS

AF REGISTERS

This pair of registers is made up of the accumulator and flag register. Except for SCF and CCF instructions the flag register cannot be manipulated directly and is used solely for checking the results of arithmetic and logical operations.

```
DEC A      ; SUBTRACT 1 FROM A REGISTER
JR NZ,TEST ; IF ZERO FLAG NOT SET JUMP
```

BASIC:

```
10. LET A = A - 1
20. IF A <> 0 THEN GOTO 400
```

The A register or accumulator as it is sometimes called, is used in most 8-bit operations. This register is used for all logical operations: AND, OR, XOR. Compare instructions also use this register and in most of these instructions the A register is assumed - CP B or CP 36. Direct memory addressing can only be accomplished by using the accumulator: LD (&4071),A.

BC REGISTER PAIR Byte Count

The BC register is used as a counter for loop operations. Register B is used as the counter in DJNZ instructions and BC in LDIR block moves or CPDR block compare, BC can also be used as a general purpose register pair performing most DE operations such as ADD HL,BC.

DE REGISTER PAIR DEstination Register

This pair of registers is second in importance only to the HL registers. They can be used in tandem with HL registers to swap data such as EX DE,HL. DE is also used in instructions such as LDIR as the destination memory pointer.

HL REGISTER PAIR High Low Register

This is the most important 16-bit register pair. It has twice as many instructions applicable to it. Immediately data can be loaded into the memory location pointed to by it and stack operations such as EX (SP),HL

THE SOURCE CHAPTER SEVEN

can easily be performed. This register pair is guaranteed to be used more frequently than any other 16-bit register pair.

IX and IY REGISTERS

Index X and Index Y

These are two 16-bit registers which are always used in this manner - all other registers can be split and used in a singular 8-bit form: A,B,C,D,E,H,L but index registers are always 16-bit except for some undocumented instructions which will be dealt with in the latter part of this book.

The index registers are often avoided by the novice but they provide a set of very powerful indexing operations which can be used with array-like structures

Once an index register has been loaded with a base pointer of memory starting address : LD IX,&4007 it is easy to manipulate values in the range -128 to +127 as offsets into an array of list.

```
LD IX,LIST
LD A,(IX + 27) ; get contents of
                ; list +27 bytes
                ; into A.
```

There is no physical difference between IX and IY and the choice is arbitrary. Index registers can be swapped with the Stack Pointer: EX (SP),IY and jumps to memory locations can also be accomplished JP (IY). Also, memory locations pointed to by these registers, along with offsets, can be directly incremented or decremented.

```
DEC (IX + 09)
INC (IY + 100)
```

One disadvantage of using these registers lies in the fact that they require an extra byte in their instruction field which does use up more memory. However, they should not be avoided because of this as they provide a very useful programming tool.

BASIC:

```
DIM A (100)
LET A = A(60)
LET B = A (20)
LET C = A (9)

LD IX, DIMA ; DIMA = table of 100 entries
LD A,(IX + 60) ; Get data from loc 60
LD B, (IX + 20) ; ETC
LD C, (IX + 09) ;
```


SP REGISTER

Stack Pointer

The Stack Pointer is a 16-bit register which cannot be separated into S and P 8-bit registers. This register plays a vital role in all assembly language programming. Most major program crashes can be attributed to misuse of this register. Any corruption of its integrity will result in the whole system, including the operating system, having a nervous breakdown.

The SP is directly affected by CALL, RET, PUSH and POP instructions and it is responsible for keeping track of the Program Counter whenever a CALL or RET from a subroutine is encountered.

The stack operates on the principle of LAST IN, FIRST OUT or LIFO. This principle is easy to grasp if you think of one of those plate stackers used in posh restaurants. Every time a plate is placed on the stack the stacker is pushed down and whenever a plate is taken off, the stacker pops up.* Obviously if you wanted to retrieve a plate half way down the stack, you would first have to take off all the plates on top of it unless you were willing to risk smashing some of the crockery.

Every time a CALL or PUSH is performed two bytes of data are pushed onto the stack. Every RET pops two bytes off the stack.

```

PUSH BC          ; PUT BC ON THE STACK
PUSH DE          ; " DE " " "
PUSH HL          ; " HL " " "

POP HL           ; PULL OFF THE STACK
POP DE           ; IN REVERSE ORDER
POP BC           ; REMEMBER LIFO

LD HL,&4007       ; PUT &4007 INTO HL REGISTERS
PUSH HL          ; SAVE HL ON THE STACK
CALL ADUP        ; CALL SUBROUTINE
POP HL           ; RETRIEVE &4007 INTO HL
<REST OF PROGRAM>

ADUP: PUSH DE     ; PUT VALUE IN DE ON STACK
      POP BC      ; GET IT OFF INTO BC
      RET         ; RETURN

```

The above program will work correctly and would preserve program integrity.

```

LD HL,&4007       ; &4007 INTO HL REGISTERS
PUSH HL          ; SAVE IT ON STACK
CALL ADUP
POP HL
<REST OF PROGRAM>

```


THE SOURCE CHAPTER SEVEN

```
ADUP: LD    DE,&100
      POP   HL
      ADD   HL,DE
      RET
```

The later example is wrong! It would result in a program crash. The integrity of the program has been corrupted and the program would try to return to address &4007 which was pushed onto the stack before the subroutine call.

* Thanks to Bill Barden Jnr. for this analogy.
In Basic every GOSUB must have a unique RETURN for the program to work correctly. In assembly language every CALL must have a RET and every PUSH must have a corresponding POP.

The Stack Pointer always builds down is memory when ever any value is pushed onto the stack.

```
LD    SP,&3000
STACK POINTER ----> &3000
PUSH  HL
STACK POINTER ----> &2FFE
PUSH  AF
STACK POINTER ----> &2FFC
POP   HL
STACK POINTER ----> &2FFE
CALL  SUBROUT
STACK POINTER ----> &2FFC
```

```
LD    SP,&3000
LD    HL,&487C
PUSH  HL
```

```
.
.
.
. . . . . STACK POINTER      &3000  STACK
.
.
.
. . . . . &2FFF  : 7C  :
.
. . . . . &2FFE  : 48  :
.
. . . . .
```

AFTER PUSH HL . STACK POINTER...: &2FFE : 48 :
AND STACK POINTER CONTAINS

PC REGISTER Program Counter

The PC register or Program Counter is a 16-bit register. This register cannot be manipulated by the programmer. It is incremented automatically to the next instruction to be executed. For instance, JP &4016 is equivalent to LD PC,&4016.

I and R REGISTERS

Interrupt and Refresh Registers

These two registers are of limited use to the programmer and their usage will be discussed later in the book.

Now we know the type of registers we have to work with we will now cover a further advantage of the Z80.

The Z80 has a very wide variety of addressing modes which can sometimes make life easier for the assembly language programmer. However, don't be frightened of addressing modes they are readily understood.

WHY DO WE NEED DIFFERENT ADDRESSING MODES?

Unfortunately it is a fact of life that all instructions cannot work with operandi that reside in the same place - if this was the case we would only require one addressing mode.

```
ADD    HL,DE      ; add contents of DE
                  ; to the contents of HL

ADD    A,(HL)      ; add the contents of
                  ; memory location pointed
                  ; to by HL to the contents
                  ; of the A register.
```

The above two instructions provide perfect examples of two different addressing modes.

IMMEDIATE ADDRESSING

In this mode the operand is contained within the instruction itself and not in a memory location.

BASIC

```
10 LET X = x + 29
20 LET Y = 3
```

```
ADD    A,29        ; add 29 to the contents of A register
ADD    A,&80        ; add 80 hex to the contents of A register
```

The statement before the comma is the instruction and the 29 is the immediate data.

Without this type of addressing and constants would have to be stored in memory addresses and this would lead to all manner of complications. We would have to keep a check on where each constant is stored and this could be very costly in processor time.

THE SOURCE CHAPTER SEVEN

```
4007 DW 21 ; store 21 in memory location &4007
4008 DW 45 ; store 45 in memory location &4008
4009 DW 128 ; store 128 in memory location &4009
```

```
LD A,(&4008) ; LOAD 45 into A register
LD B,A ; put it into B
LD A,(&4009) ; LOAD 128 into A
ADD A,B ; Now ADD 45 contained in b
; to the 128 contained in A
; and store the result in A
```

BASIC

```
10 POKE &4007,21
20 POKE &4008,45
30 POKE &4009,128
40 LET A = PEEK (&4008)
50 LET B = A
60 LET A = PEEK (&4009)
70 LET A = A + B
```

IMPLIED ADDRESSING

This addressing mode is used for simple instructions that do not require operandi and the register is not named in the mnemonic Eg.

```
SCF ; Set Carry Flag
CCF ; Complement Carry Flag
NOP ; No Operation
EI ; Enable Interrupts
```

The above are just a few of the instructions that fall into the implied addressing mode. Because these instructions specify a simple operation they do not require an operand and are one byte instructions.

In simple terms, the CPU knows that when it encounters an instruction such as SCF it will set the carry flag and realise that there is no need to fetch any more data for that instruction.

DIRECT ADDRESSING

BASIC

```
10 LET X = PEEK (&4007)
```

```
LD A,(&3EFA) ; load A with the contents
; of memory location 3EFA
LD (&4007),A ; store the contents of A
; in memory location 4007
```


BASIC

```
10 LET H = 239
20 LET L = 69
30 POKE 16391,L
50 POKE 16392,H
```

```
LD HL,&EF45 ; LD H with 239 and L with 69
; Think about it!
LD (&4007),HL ; Store &45 in 4007 and
; &EF in &4008
```

As we have already discussed, the Z80 stores its values Least Significant Byte/Most Significant Byte (LSB/MSB) so the instruction:

```
LD(&4007),HL
```

will result in &4007 containing &45 and location &4008 containing &EF.

Another group of instructions that make use of the direct mode are the call, and jump instructions.

BASIC

```
20 GOTO 100
40 GOSUB 300
```

```
JP 100
CALL 300
```

REGISTER ADDRESSING

This addressing mode allows single 8-bit registers to communicate with each other or with the A register and is used by such instructions as:

```
ADD A,C
OR
ADD IY,SP
```

It can also be used with the 16-bit register pairs as shown is the last example above.

REGISTER INDIRECT ADDRESSING

This is quite a useful addressing mode, as we shall see in a moment, and is a 'hand me down' from the older 8008. The later 8008A added the capability to use BC and DE as 'pointers' for loading or storing the A register.

THE SOURCE CHAPTER SEVEN

Because many of the Z80 instructions do not allow the operand to be addressed directly, this instruction was carried forward. For instance, while it is possible to load the A register directly from memory:

```
LD A,(&4007)
```

It is not possible to : ADD A,(&4007). However, it is possible to point the HL register at a memory location and then do a variety of other things. As you may now have realised, the A register is the only register (with a couple of exceptions involving the HL registers) which can load or store directly to memory addresses. All other registers must use indirect methods to load or store data.

Let's say that we wanted to load B and C registers with the contents of memory locations. Well, there are two ways of doing this which are highlighted in the following examples.

INDIRECTLY VIA THE A REGISTER

BASIC

```
10 LET A = PEEK(16391)
20 LET B = A
30 LET A = PEEK(16392)
40 LET C = A
```

```
LD A,(&4007) ; load A from memory location 4007
LD B,A       ; store contents of A in B
LD A,(&4008) ; do the same with
LD C,A       ; C register
```

INDIRECTLY USING THE HL REGISTERS

BASIC

```
10 LET HL = 16391
20 LET B = PEEK(HL)
30 LET HL = 16392
40 LET C = PEEK(HL)
```

```
LD HL,&4007 ; Point HL at memory location 4007
LD B,(HL)   ; Load B from memory location
             ; pointed to by HL
LD HL,&4008 ; Point HL at memory location 4008
LD C,(HL)   ; Get value from memory location
             ; pointed to by HL into C
```

RELATIVE ADDRESSING

This mode of addressing is used for relative jump instructions; NO OTHER INSTRUCTIONS USE THIS MODE.

JR NZ, LOCATION

Relative addressing allows a jump within a limited range of 256 bytes. That is to say, we can alter the program from relative to the PC (Program Counter) within the range -128 through to +127.

The JR instruction saves one byte over the normal JP statement and since most program jumps will fall within this range it is a very valuable instruction.

INDEX ADDRESSING

This is a powerful addressing mode. It allows you to retrieve or store data from tables set up in memory. We can make the IX or IY registers point to an address then add an offset within the range of -128 to +127. If the IX register pointed to memory address &4007 we can the LD A,(IX + 15) which would load the A register with the contents of memory location &4016 and LD A,(IX + 00) would load the A register with the contents of memory location &4007.

BASIC

```
10 DIM IX (15)
```

```
20 LET A = IX (15)
```

```
4007 DW 100 ; Make memory location 4007 = 100
4008 DW 36 ; etc.
4009 DW 80
400A DW 55
```

```
LD IX,&4007 ; Point IX at memory location 4007
LD A,(IX+02) ; Load A from &4009
; A now = 80
LD A,128 ; Move 128 into A register
LD (IX+03),A ; Change memory location
; 400A from 55 to 128
```

PAGE ZERO ADDRESSING

This mode allows one byte instructions known as RST CALLS to be used in place of the normal CALLS to subroutines. However, only the first page of memory is allowed for these instructions. The Einstein's ROM makes excellent use of this type of CALL.

```
RST10 ; Make a call to subroutine at
; memory location &10
```

That just about sums up the addressing modes of the Z80 CPU. Worry not over the technicalities, addressing modes really do improve life for the assembly language programmer.

REMEMBER DATA FLOWS FROM RIGHT TO LEFT

Think of the LD mnemonic as the Basic command LET : -

```
LD A,10      .... LET A = 10

LD HL,35126   .... LET HL = 35126
LD A,(HL)     .... LET A =PEEK(HL)

LD B,10       .... FOR B = 0 TO 10
LD HL,35412   .... LET HL = 35412
LD DE,7000    .... LET DE = 70000
LD B,10       .... FOR B = 1 TO 10
LOOP: LD A,(HL) .... LET A = PEEK(HL)
LD (DE),A     .... POKE DE,A
DJNZ LOOP     .... NEXT
```


CHAPTER EIGHT

There are two methods of writing machine code programs : the hard way or the easy way. The hard way is to hand assemble each instruction into hexadecimal and poke it into memory. This method is very tedious and prone to mistakes. The easy way is to allow the computer to deal with translating the code, and the program that allows the micro to do this is called an assembler.

The assembler translates our code, written in mnemonic instructions (LD, LDIR etc.) and called the source file, into the object code, a machine language program which the computer executes when loaded into memory.

Input into assembler	----->	SOURCE CODE
Output from assembler	----->	OBJECT CODE

The assembler requires an Ascii file of mnemonic instructions, and almost any text editor will do, I use Wordstar. If any errors are found, at assembly time, it is a simple matter to re-enter the text editor and correct the mistakes.

There are many assemblers on the market and the choice depends upon the amount of money available, and how frequently you intend to use it. My personal preference is Macro 80 by Microsoft which is an excellent assembler but is quite expensive and does not supply a text editor. Hi-soft's DEVPAC80(version 2) is excellent value for money and supplies a text editor plus a very versatile debugger. Because Devpac80 is a "middle of the road" package all future examples, in this book, will be constructed with this package, and any poking into memory will be with Promon, the

THE SOURCE CHAPTER EIGHT

debugger - yes, I know I could use Mos, but I prefer Promon.

It is a certain bet that 90% of assembled machine code programs will be "bugged". Even a simple, ten line program may contain one or two nasties. Some bugs are obvious, and can be traced simply by scanning the source code. However, the majority of bugs are not easy to spot, and one wrong byte can send the computer on that journey to nowhere and the only recourse is to switch off and start again. This is the time that the debugging program comes in very handy.

A debugger allows us to load in the object program and single step through each instruction. At each stage we can examine the Z80 registers to validate the data they hold. If we are lucky enough to own an eighty-column card and a green screen monitor, we can also watch the build up of a graphic screen while single stepping. Break points can be inserted into the program. A break point simply stops the program at a particular place and allows us to check various data in the program. The only way to become familiar with a debugger is to "mess with" all its features.

O.k. We have all got our assemblers handy so let's get back to our original intention of adding a new command to Basic by writing a short routine and installing it, permanently, in the Interpreter.

Most versions of Basic have a facility for locating a variable in memory. This facility is normally called Varptr (variable pointer). This is a good place to start, for reasons that will become apparent at a later stage.

Our new command will be VARPTR(n) where n represents the variable (simple, dimensioned or string) that we want to locate in memory.

```
LET A = 3
LET X = VARPTR(A)
PRINT X
```

This program will return the address where the data (3), relating to the variable A, is stored. If the variable is a string it will return the starting address of where the actual string data is stored.

```
LET X$ = "ABC"
LET X = VARPTR(X$)
FOR I = 0 TO 2
PRINT PEEK(X)
NEXT I
```

The above short routine would print ABC.

STEPS REQUIRED TO ADD A NEW COMMAND

- a] MAKE A BACKUP COPY OF XBAS ON A BLANK DISC AND ONLY USE THIS BACKUP NOT THE ORIGINAL !

- B] The AUXILIARY RESERVED WORD TABLES must be moved to allow them to be expanded.
- c] The AUXILIARY ADDRESS TABLES must also be moved to allow the address of the new command to be included.
- d] PTR 0 The start of Basic must be changed to reflect the changes ... Basic will start higher in memory after the new command is added.
- e] PTR 3 & PTR 8 must be changed to point to the new addresses where the two tables now reside.
- f] The very first byte in the Auxiliary Word tables must be incremented by one to include our new command.

Remember, we are going to modify Basic and do not want any mishaps .. do not, under any circumstances, use your original Xbas file.

First, we must write our new routine. Enter your editor and type in the following listing. You can leave out the comments if you wish. When you have checked that your version tallies exactly with the listing below, save it to disc.

```
2:
;VARPTR ... FIND THE ADDRESS OF A BASIC VARIABLE AND RETURN IT
;FOR USE IN EINSTEIN BASIC V4/256
;
;      org      3bc5h      ;This is were we're going to put it.
;
;Equates from basic here .....
;
getvar equ      18B3H
vrtype equ      0166h
convert equ      04dah
exit   equ      0290h
```


THE SOURCE CHAPTER EIGHT

```

varptr:
    pop    hl          ;Get pointer into Basic line
    inc    hl          ;bump to variable
    call   getvar      ;Get variable and put variable type into
                        ;vrtype.
    push   hl          ;save basic line position
    ld     a,(vrtype)
                        ;Check type of variable
                        ;0 = simple or array variable
                        ;1 = full array etc
    or     a           ;check if = 0: or a sets Z flag if = 0
    ex     de,hl       ; returns from last call with DE pointing
                        ;to contents

    jr     z,fpa       ;not a string

    inc    hl
    inc    hl          ;Skip past pointers and get actual string
                        ;address .. hl now points to it.
    ld     a,(hl)      ;Lsb of address
    inc    hl          ;now align so we can get
    ld     h,(hl)      ;Msb of address
    ld     l,a         ;hl now holds actual address
    xor    a           ;must always zero variable type
    ld     (vrtype),a  ;before exiting

fpa:
    call   convert     ;Convert into a fpa number in range 0-65535
    jp     exit        ;All done so exit through correct path in
                        ;order that all pointers are correct when
                        ;returning to Basic.

end

```

Now type : GEN80 VARPTR;LIST+,W

This will assemble the file and save a PRN (print file) to disc. We now need to print this file to extract the machine code bytes that we are going to insert into the Basic Interpreter. If GEN80 has assembled the file, and you have made no typing mistakes, you should receive Pass 1 & Pass 2 "no error" message. Now make a listing of the PRN file which should look like the following : -

Pass 1 errors: 00

```

0100      1 ;VARPTR ... FIND THE ADDRESS OF A BASIC VARIABLE AND RETURN IT
0100      2 ;FOR USE IN EINSTEIN BASIC V4/256
0100      3 ;
3BC5      4      org      3bc5h      ;This is were we're going to put it.
3BC5      5 ;
3BC5      6 ;Equates from basic here .....
3BC5      7 ;
3BC5      8
18B3      9 getvar      equ      18B3H
0166     10 vrtype      equ      0166h
04DA     11 convert      equ      04dah
0290     12 exit        equ      0290h
3BC5     13
3BC5     14
3BC5     15
3BC5     16 varptr
3BC5 E1   17      pop      hl      ;Get pointer into Basic line
3BC6 23   18      inc      hl      ;bump to variable
3BC7 CDB318 19      call     getvar    ;Get variable and put variable type into
3BCA      20                      ;vrtype.
3BCA E5   21      push     hl      ;save basic line position
3BCB 3A6601 22      ld       a,(vrtype)
3BCE      23                      ;Check type of variable
3BCE      24                      ;0 = simple or array variable
3BCE      25                      ;1 = full array etc
3BCE B7   26      or       a        ;check if = 0: or a sets Z flag if = 0
3BCF EB   27      ex       de,hl     ; returns from last call with DE pointing
3BD0      28                      ;to contents
3BD0      29
3BD0 280A 30      jr       z,fpa     ;not a string
3BD2      31
3BD2 23   32      inc      hl
3BD3 23   33      inc      hl      ;Skip past pointers and get actual string
3BD4      34                      ;address .. hl now points to it.
3BD4 7E   35      ld       a,(hl)    ;Lsb of address
3BD5 23   36      inc      hl      ;now align so we can get
3BD6 66   37      ld       h,(hl)    ;Hsb of address
3BD7 6F   38      ld       l,a      ;hl now holds actual address
3BD8 AF   39      xor      a        ;must always zero variable type
3BD9 326601 40      ld       (vrtype),a ;before exiting
3BDC      41 fpa
3BDC CDDA04 42      call     convert    ;Convert into a fpa number in range 0-65535
3BDF C39002 43      jp       exit      ;All done so exit through correct path in
3BE2      44                      ;order that all pointers are correct when
3BE2      45                      ;returning to Basic.
3BE2      46
3BE2      47      end

```

Pass 2 errors: 00

Symbol Table used: 00K out of 16K.

THE SOURCE

CHAPTER EIGHT

The format of the print file is :

Address	Machine code	Instruction	Line number	Mnemonic	Comment
---------	--------------	-------------	-------------	----------	---------

The numbers that are of interest to us are the actual machine code bytes in column 2 starting at line number 17 (address &3BC5).

```
E1,23,CD,B3,18,E5,3A,66,01,B7
EB,28,0A,23,23,7E,23,66,6F,AF
32,66,01,CD,DA,04,C3,90,02
```

All Hexadecimal

We are now going to take these numbers and insert them into Basic by using PMON - if you don't possess Devpac80 you can use the Mos functions... see your Mos Manual for details.

Make sure you have Promon in Drive 0 & Copy of Xbas in Drive 1.

```
Type :- PMON <RET>      ; Pmon loaded
          F <RET>        ; File access required
          R              ; Want to load a file
          B:XBAS         ; Filename is Xbas
          100            ; Starting address

          MA             ; Memory address .. we are going to modify
                        ; PTR 3 to point to new location of Aux Word Tab
          2B88           ; Address of PTR 3

          MW             ; Go into window mode so we can edit
```

; You should now see ">" on the left of the screen pointing to &2B88

```
Type :- 013E           ; Numbers store Lsb/Msb = 3E01. Check to see
                        ; if the locations hold these two numbers.
```

;PTR 3 now points to &3E01 and this is where AUX WORD TABLE is going

```
Type :- ESC            ; Exit window mode
          MM            ; Memory move
          3BC5          ; First
          3C34          ; Last
          3E01          ; where
```

;Table now moved to &3E01

```
Type :- MA             ; Memory Address
          3E7F          ; End of the table just moved.
          MW            ; Memory Window so we can edit.
```

; You should now see in the memory widow ">" pointing to 80. This tells the interpreter that it is the end of the table. We are now going to

;insert our new command Varptr. Remember, the first byte of the word is the Ascii number plus &80 ... this tells the computer that it is the start of a new word.

V = ascii &56 + &80 = D6

A = " &41

R = " &52

P = " &50

T = " &54

R = " &52

(= " &28

Type :- D6 41 52 50 54 52 28

; New word now installed you should see ARPTR
; in the extreem left hand column.

Type :- 80
ESC
MA
3E01

; Insert new end of table marker.
; Exit window mode
; Memory address
; This is the very first byte in this table it
; should read &19 which is the number of
; commands already installed .. we need to add 1
; Memory window
; We are in Hex so 19 +1 = 1A ! = 26 decimal
; Exit memory window

; We now need to move the AUX address table. We are going to move it a few
; bytes hihger than the end of the AUX Word Table so we still have a bit of
; room to add another command ... if we wish.

Type :- MM
3C35
3C66
3E80

; Memory move
; First
; Last
; Where

; The AUX word table is now re-located. We now have to install the address
; where are Varptr will be situated. Because there is now some empty space
; where the tables used to be we can utilise this because our routine is
; doesn't take up that much memory.

Type :- MA
3EB2
MW
C53B
ESC

; Memory address
; This is the first empty byte in the AUX addr
; table should contain FF or 00
; Memory window to edit
; Lsb/Msb 3BC5 where our Varptr is going
; Exit memory window

; Now we change PTR 8 to reflect the new address of the AUX word table
; Otherwise Basic wouldn't be able to find it !

Type :- MA
2B92
MW

; Memory address
; PTR 8
; Memory window to edit

; The ">" should be pointing to 353C buyes in the left hand window

THE SOURCE
CHAPTER EIGHT

```
Type :- 803E      ; Lsb/Msb 3E80 Start of new AUX word Table
          ESC      ; Exit memory window
```

; We can now clear out the old table address so that we can put our routine
; in the vacant space

```
Type :- MF      ; Memory fill
          3BC5    ; First
          3C66    ; Last
          0       ; With ... filled with zero
          MA      ; Memory address we now change PTR 0 to new
                  ; Start of Basic ... to keep it neat our new
                  ; Start will be $4100
          2B84    ; PTR 0 start of Basic pointer
          MW      ; Memory window for edit
          0041    ; Lsb/msb = &4100
          ESC
```

; Now it is time to punch in our routine and see if it will work ??

```
Type :- MA      ; Memory address
          3BC5    ; This is where we start
          MW      ; Memory window
```

; Right, start typing those machine code instructions into memory

E123CDB318E53A6601B7 etc

```
ESC      ; Exit memory window
```

; O.k. That's it. We now have to save it to disc. We will call this new
; file X1 .. this way, if you have made any mistakes, you can still use ;
your copy of Basic to try again hopefully you will have got it ; right
first time !!

```
Type :- F      ; File access
          W      ; We want to write a file !
          B:X1.com ; Write file named X1.com to Drive 1
          100    ; First byte
          4100   ; Last byte
```

; Disc switches on and saves your new version of Basic called X1.COM

Right. Let's try this new Basic. Re-set the computer and load X1. You
should now have Basic loaded into memory and be left with the input prompt.
Try this short program : -


```
10 LET A = 3
20 LET X = VARPTR(A)
30 PRINT X
40 FOR I = 0 TO 3
50 PRINT HEX$(PEEK(X+I));" ";
60 NEXT I
```

You are now looking at the way in which the value , assigned to a variable, is stored in memory.

0000 0000 0040 0082

It is very difficult to ascertain how floating point numbers are stored, but as far as I can figure out, and I may be wrong, the exponent is calculated as a power of two with the high bit set if the number is positive. the mantissa is stored in the lower three bytes with the top bit equal to zero if positive and set if it is negative. I will leave you to experiment with this problem - the fact is, it works. The Einstein is reasonably accurate with floating point numbers.

```
A = 1    = 00 00 00 81  = 2Δ0
A = 2    = 00 00 00 82  = 2Δ1
A = 4    = 00 00 00 83  = 2Δ2
```

The reason we installed Varptr will be explained in a short while but it certainly wasn't done with the intention of having a brain haemorrhage trying to sort out floating point numbers.

```
10 LET X$ = "ABC TEST"
20 LET X = VARPTR(X$)
30 FOR I = 0 TO 7
40 PRINT CHR$(PEEK(X));
50 NEXT
```

The above program will print out :- ABC_TEST.

If we wanted to install a machine code subroutine into our Basic program we would first have to reserve memory for the routine with the CLEAR command. Also, we would have to calculate how much memory we require, and after installing the routine we would have to make sure that Basic never overwrites the subroutine. Most subroutines are very small by comparison with the Basic program, and it would be nice to have some other method of storing these routines. Well, just use your imagination - with Einstein's Basic almost anything is possible.

THE SOURCE CHAPTER EIGHT

A CHOICE REMARK

Type in the following Basic program exactly as listed below. Save it to disc before attempting to run it ! This is a unique method of storing machine code programs. This will support up to 255 bytes of machine code. There is no need to reserve memory and Basic will never interfere with the routine.

The rem statement must be set up exactly to the number of bytes in the subroutine.

```
10 REM ***** = 13 bytes of machine code
```

After you have run the program you will not be able to list it as the Basic line has been altered and so make sure you always save the program first, otherwise it is a complete re-type.

This is only one way of storing machine code routines. And if you take time to analyse how Basic operates you should be able to come up with the answers.

```
10 REM *****
20 FOR I = &3E17 TO &3E17+49
30 READ A: POKE I,A
40 NEXT
50 INPUT X
60 CALL &3E17
70 GOTO 50
80 DATA &01,&0F,&06,&CD,&42,&3E,&01
90 DATA &47,&07,&CD,&42,&3E,&01,&10
100 DATA &08,&CD,&42,&3E,&01,&10,&09,&CD
110 DATA &42,&3E,&01,&10,&0A,&CD,&42
120 DATA &3E,&01,&10,&0C,&CD,&42,&3E
130 DATA &01,&08,&0D,&CD,&42,&3E,&09
140 DATA &7B,&03,&02,&79,&03,&03,&09
```

Ready

Now, let's get to our Varptr routine and put it to some useful purpose within one of our Basic programs.

Storing graphics within a string is a tedious method. All those LET X\$ = CHR\$(23)+CHR\$(56)+Y\$+ etc. etc. However, this is the only way we can display compound graphics without building up complex sprite patterns. Our Varptr routine will allow us to simplify the matter and data can be "plugged" into the string simply by using POKE. Using this method allows us to change any one character without having to split strings and work out complicated string manipulation routines. Study the following Basic program, then type it in and run it.

```

9 REM INITIALISE STRING
10 X$="123456789ABCD"
19 REM TEST IT
20 PRINT X$
29 REM GET ADDRESS OF STRING IN MEMORY
30 LET A = FPTX$)
39 REM NOW CHANGE STRING DATA
40 FOR I= 0 TO 12
50 READ D
60 POKE A+I,D
70 NEXT :CLS
80 PRINT @0,10;X$;
90 LET A = FPTX$)
100 C=PEEK(A+3)
110 POKE A+3,&EA

```

This is only a very simple example and really doesn't do very much but, I am sure, you will be able to utilise this concept and use your own ideas to construct some very useful routines. Notice that the LET statement is mandatory when using Varptr this is because of the exit address used when interfacing it to Basic.

THE SOURCE CHAPTER EIGHT

Whenever you use the string function, as above, you must always call the varptr routine before altering the string contents. This is due to the fact that string space is dynamic and as you add or delete strings, the actual address of strings will change accordingly.

If you experiment with arrays you should realise that there is a rather unique way machine code programs may be stored and data fed to the routine via elements of the array. This is a path you can examine when you feel in an adventurous mood.

CHAPTER NINE

THE VIDEO DISPLAY

All display operations are managed by the Texas TMS9129 Video Display Processor (VDP). The chip is not a secret weapon developed by Texas Instruments in order to fill up the psychiatric wards with budding programmers. It is a sophisticated piece of electronic wizardry which allows complex screen displays to be utilised. However, as with all powerful electronics, the chip requires what appears, at first sight, a complicated set of instructions. The VDP manages an area of ram, which is extra and extra to normal ram called Video Ram (VRAM).

The VDP communicates with VRAM via Ports 8 and 9.

Port 9 is used for address transfers
Port 8 is used for data transfers

All addresses throughout VRAM are 14-bit. Address transfers require a two-byte transfer with 2 bits unused.

The VDP has five available display modes.

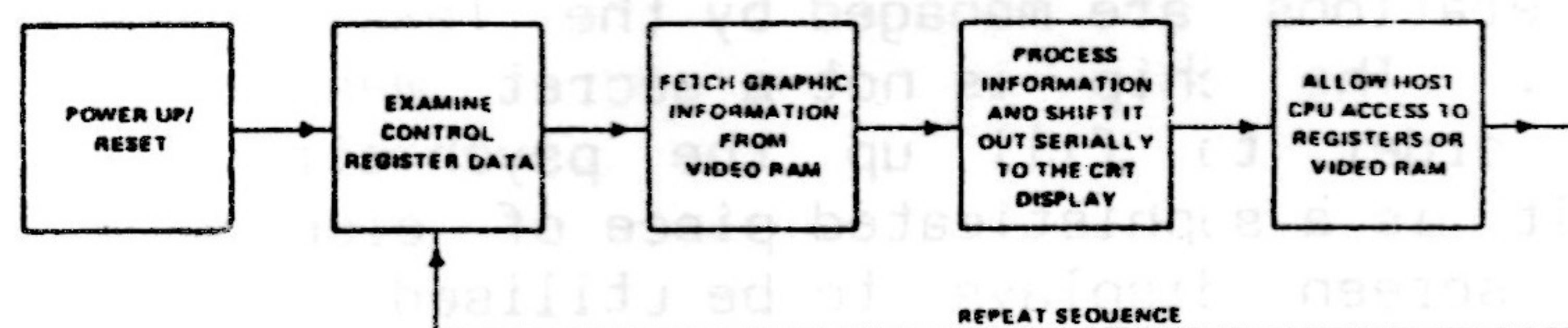
- a) TEXT
- b) MULTICOLOUR MODE
- c) GRAPHIC MODE 1
- d) GRAPHIC MODE 2
- e) MODIFIED GRAPHIC MODE 1

TEXT MODE provides a screen which is 40 columns wide by 24 rows deep. Two colours are available in this mode.

GRAPHIC MODE 2 offers a display of 32 columns by 24 rows deep. Sixteen colours are available and plotted displays are also allowed.

OVERVIEW

The VDP patches data from video ram and, after processing, the data is used to control the beam of the CR Tube as it scans the screen. This sequence is repeated over and over again similar to an endless FOR/NEXT LOOP. The chip also performs many other functions such as taking time out to check if the Z80 chip requires access to VRAM or any of the VDP's internal registers.



The VDP has nine internal registers. Eight registers contain control bits which may be programmed by the user. The ninth register is the STATUS REGISTER and this may be interrogated to determine what is taking place within the VDP.

The dedicated VRAM is mapped from #0000 through to #4000 and is used to store the various tables associated with screen data, colour data, character data and sprite data. The tables that hold these data are not fixed, and with a few exceptions, can be placed anywhere in VRAM by the programmer.

In practice it is also possible to store program data in the unused portions of VRAM thus saving normal ram.

The start of each VRAM table is arbitrary and is determined by what values are placed in the eight write only VDP registers. For many programmers this is their first stumbling block.

As already stated, the EINSTEIN communicates with the VDP through Ports 1 and 2 and the Z80 can be programmed to perform one of the following operations.

WRITE ONE BYTE OF DATA TO VRAM
READ ONE BYTE OF DATA FROM VRAM

WRITE TO A VDP INTERNAL REGISTER
SET UP A VRAM ADDRESS
READ THE VDP INTERNAL STATUS REGISTER

PORT EIGHT [OUT(08) : IN (08)] is used for data transfers.
PORT NINE [OUT(09)] is used for address transfers.
PORT NINE [IN (09)] is used to read the VDP status register.

MODES

Under assembly conditions five modes are available, although I can see no use for the MULTICOLOUR MODE.

TEXT MODE provides a screen 40 columns by 24 rows in two colours.

MULTICOLOUR MODE allows 64 times 48 colour dot display in fifteen colours and thirty-two sprites available.

GRAPHICS MODE 1 provides 256 by 192 pixel display in fifteen colours, transparent with thirty two sprites.

GRAPHICS MODE 2 is an enhancement of Graphic Mode 1 allowing more complex colour and pattern displays with thirty two sprites.

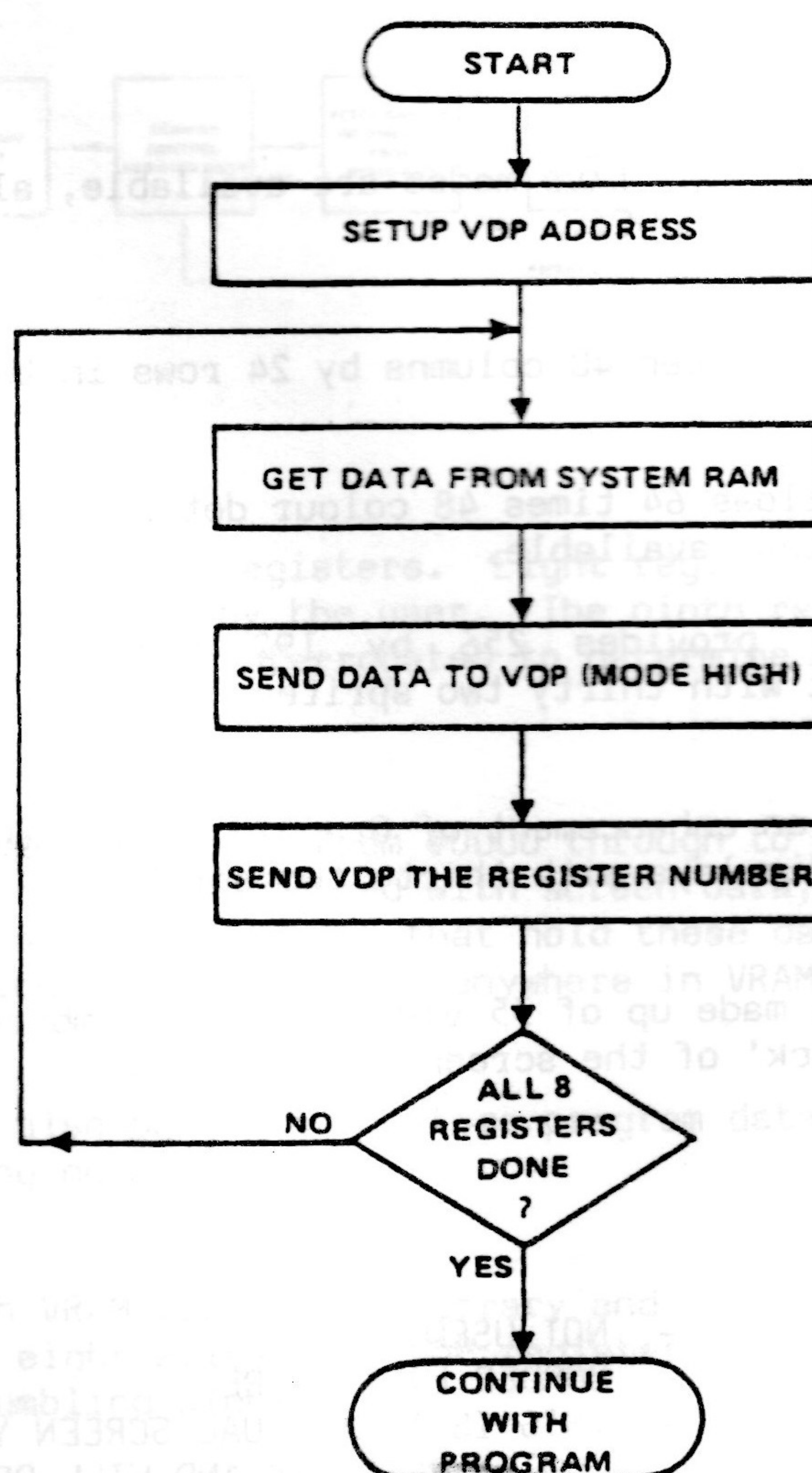
The Video Display is made up of 35 video planes numbered from 34 down to 0, working from the 'back' of the screen to the front.

PLANE

NO.		
34	EXTERNAL VIDEO	- NOT USED
33	BACKDROP	- BORDER LINES HERE
32	PATTERN	- THIS IS THE ACTUAL SCREEN YOU PRINT
31	SPRITE	- LOWEST PRIORITY AND WILL BE HIDDEN BY ANY OF THE OTHER SPRITES
30-1	SPRITES	
0	SPRITE	- HIGHEST PRIORITY SPRITE WILL APPEAR TO PASS IN FRONT OF ANY OF THE OTHER SPRITES

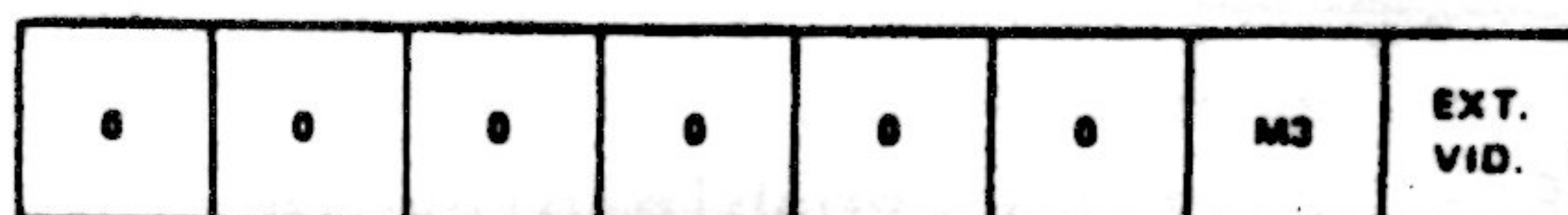
No sprite planes are allowed in TEXT MODE.

MODES are set by using a combination of bits in VDP Register 0 and 1 and before we can use any mode the VDP must be initialised. The diagram below illustrates the steps involved in initialising the VDP Registers.



REGISTERS

REGISTER 0

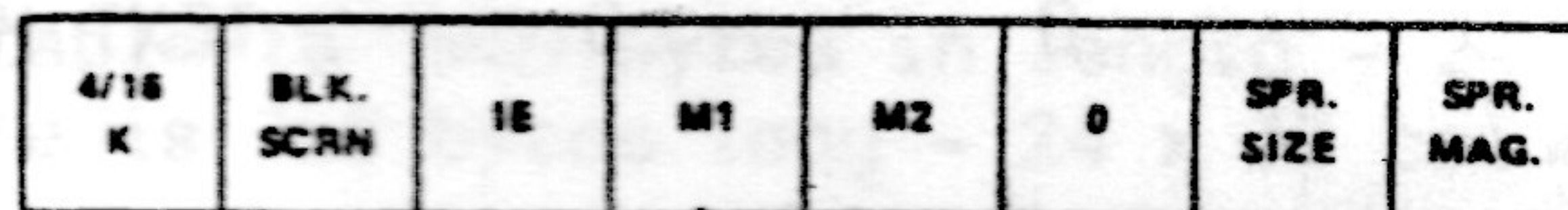


THIS REGISTER CONTAINS TWO VDP CONTROL BITS

This register contains two VDP control bits, other bits are reserved for future use and must always be set to zero.

BIT 0 EV - EXTERNAL VIDEO ENABLE/DISABLE
1 enables external video
0 disables external video

BIT 1 M3 - MODE BIT 3. This is one of three bits which determine the display mode in which the VDP will operate. The other MODE bits (M1 + M2) are in register one.



REGISTER ONE CONTAINS EIGHT CONTROL BITS

REGISTER 1 - contains seven VDP option control bits. BIT 2 is reserved for future expansion and must be set to Zero.

BIT 7 - is the 4/16K toggle for RAM selection.
1 = 4108/4116 RAM
0 = 4027 RAM

On the EINSTEIN it is always set to 1. as the computer uses 4116 Ram chips for VRAM.

BIT 6 - BLANK ENABLE/DISABLE
0 = disable active display
1 = enable active display

When set to Zero the screen will show border only and the display area will be blanked. This is useful for blanking out sprites and the backdrop plane at a stroke. Blanking by this method does not destroy any of the tables in VRAM.

BIT 5 - IE = Interrupt enable/disable
0 = disables interrupts
1 = enables interrupts

How this can be used to the best advantage is explained in a later chapter.

BIT 4 - PATTERN MODE BIT M1

BIT 3 - PATTERN MODE BIT M2

These two bits, used in conjunction with M3 (bit-1) in register Zero set up the display to operate in one of the following modes.

M1	M2	M3	
0	0	0	- GRAPHIC MODE 1
0	0	1	- GRAPHIC MODE 2
0	1	0	MULTICOLOUR MODE
1	0	0	TEXT MODE

BIT 2 - Reserved for future expansion
Must be Zero

BIT 1 - Selects the size of Sprites

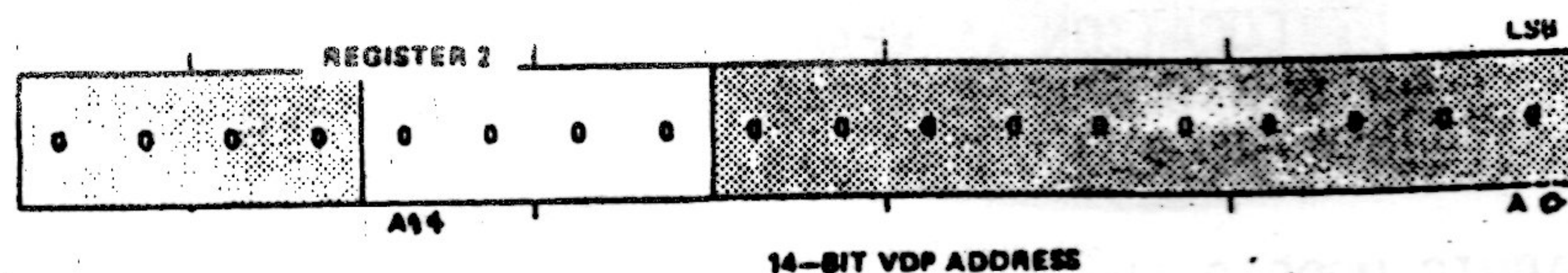
0 = 8 x 8 bit sprites
1 = 16x16 bit sprites

BIT 0 - Selects sprite magnification

0 = No Magnification
1 = Magnify by two

Thus 8 x 8 sprites become 16x16 and 16x16 become 32x32

REGISTER 2



This register tells the VDP where the starting address of the name table (screen) sub block is located in VRAM.

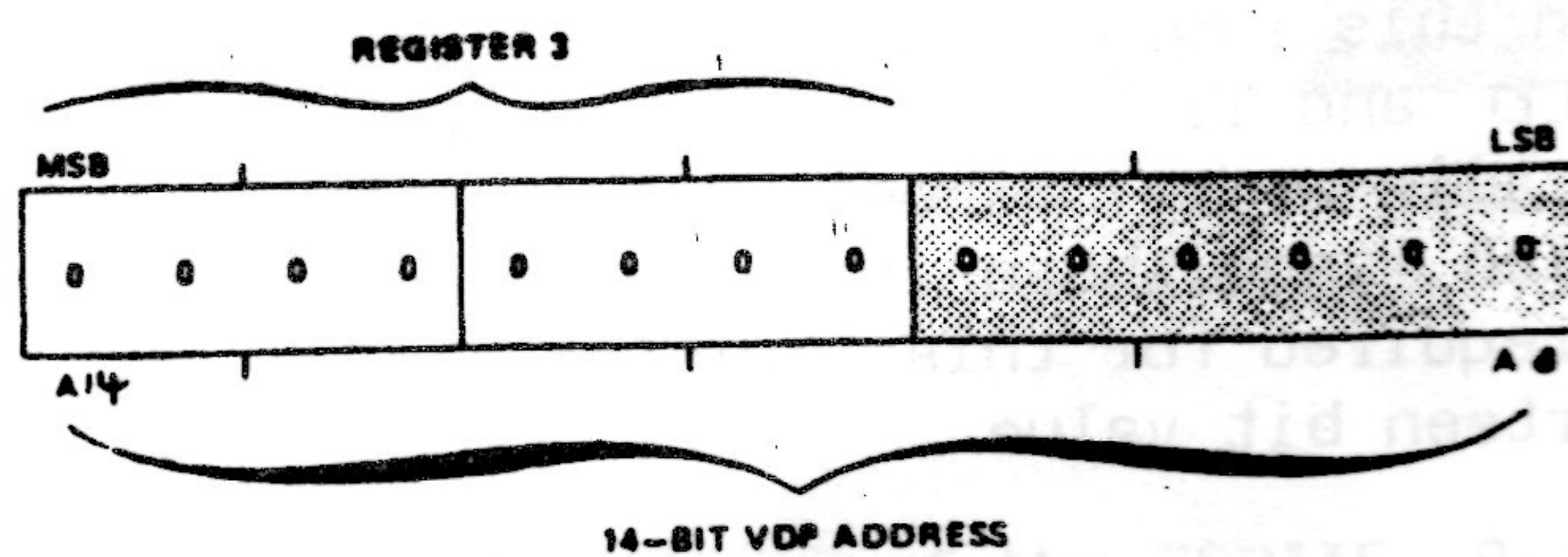
Only four bits are used by the programmer so the range of values that can be sent to this register lie between 0 and 15.

START ADDRESS OF SCREEN - CONTENT REG 2 * 1024

It is therefore obvious if we loaded this register with 2 the screen would start at 2048 (800H). Inversely, if you wanted the screen to start at 6144 (1800H) then just divide the desired location by 1024 (400H) = $1800H / 400H = 6$. You would send 6 to this register.

In the Text Mode this table is 960 bytes in length - 24 x 40 columns. In all other modes the table is 768 bytes long - 24 x 32 columns.

REGISTER 3



The contents of this register define the starting addresses of the COLOUR TABLE. This table holds the colours of the patterns used in GRAPHIC MODES 1 and 2 and is not used in the TEXT MODE.

Graphic Modes 1 and 2 utilise the colour table in different ways which means that the set up varies dependant on the operating mode.

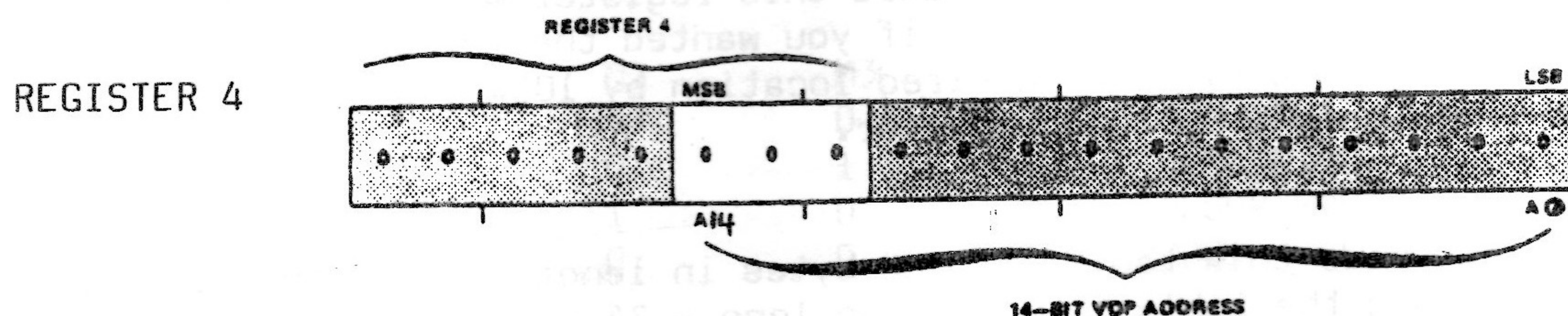
In GRAPHIC MODE 1 the colour table is located on a 64K boundary and is 32 bytes in length.

LOCATION IN VRAM = CONTENTS REG 3 X 64
VALUE FOR REG 3 = DESIRED LOCATION/64

In GRAPHIC MODE 2 the table is 6144 bytes long. The table must be located on an 8K boundary. Because the VDP uses 16K of VRAM this means that the COLOUR TABLE can either start at 0000 or 8192 (2000H).

If you want to locate the Colour Table at Zero then bit-7 of Register 3 must be 0. To locate the table at 8192 set bit-7 to 1 and in both instances bit 6,5,4,3,2,1,0 must be 1's.

COLOUR TABLE LOCATION	VALUE FOR REG 3
0000	127 (7FH)
8192 (2000H)	255 (FFH)



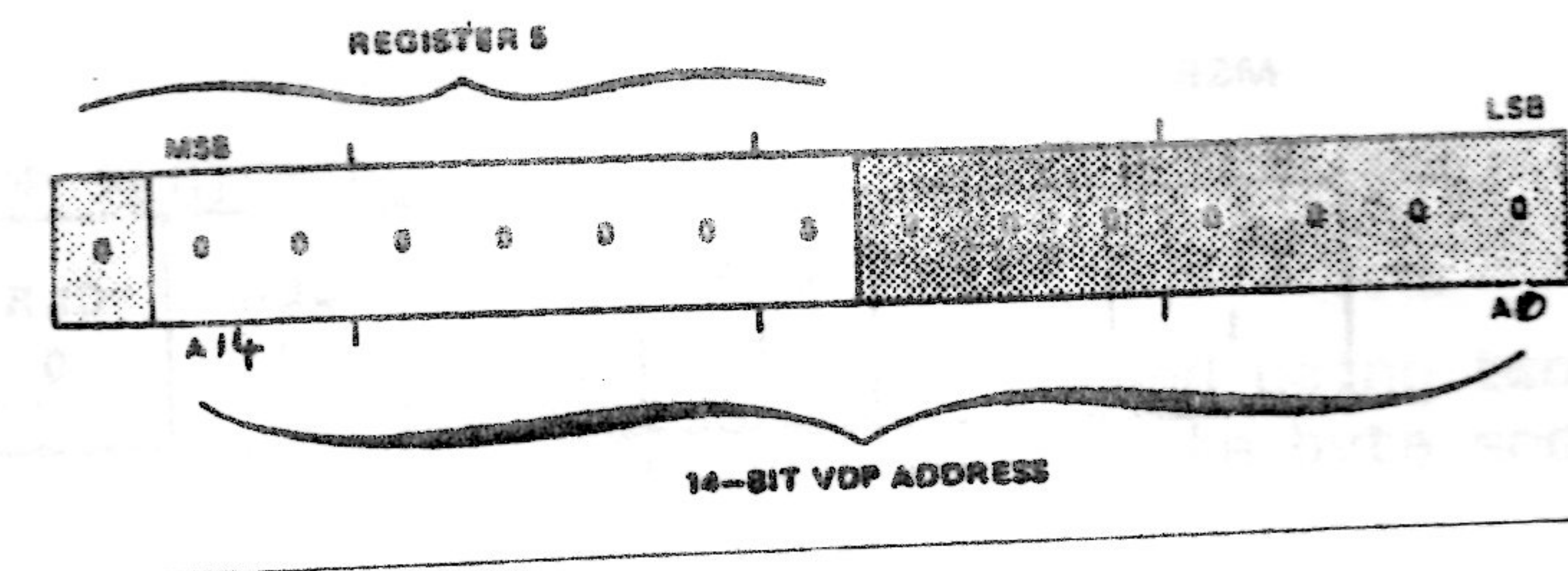
The value in this register defines the starting location of the PATTERN GENERATOR TABLE and is obviously where the patterns for display are held in memory. In all modes except GRAPHIC MODE 2 it is located on 2K byte boundary, which means there are eight possible locations. Only three bits of data are required for this register and the data forms the upper three bits of a fourteen bit value.

DESIRED LOCATION/800H = CONTENTS REG 4

In GRAPHIC MODE 2 the table must be located on an 8K boundary - Bits 1 and 0 must be 1. If bit-2 is set to 1 the table will be located at 8192 (2000H) and if it is 0 the table will be located at 0.

PATTERN GENERATOR LOCATION	VALUE FOR REG 4
0000	3
8192 (2000H)	7

REGISTER 5



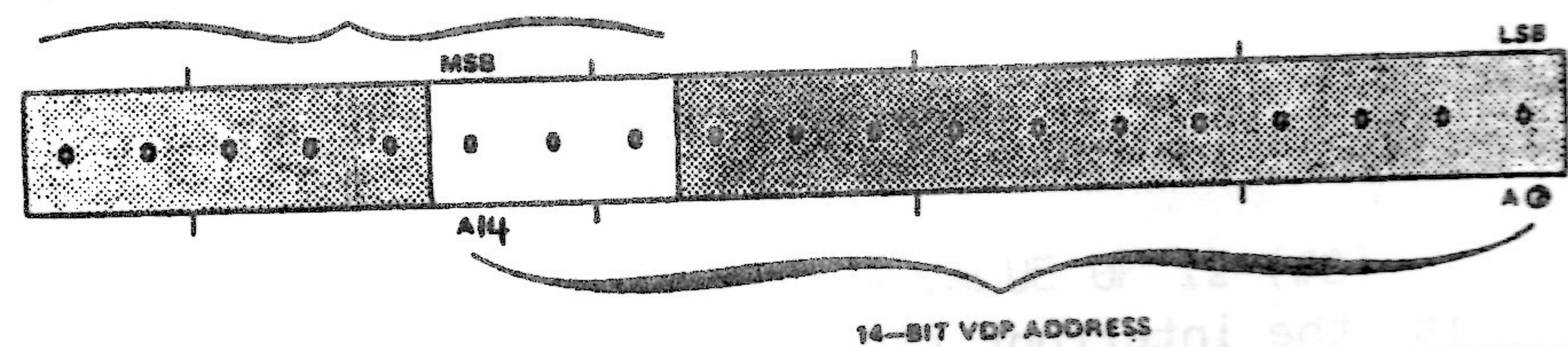
The value of this register places the SPRITE ATTRIBUTE TABLE starting address. It is a SEVEN BIT register and must be located on a 128 byte boundary.

The table is 128bytes in length and describes position, colour and pattern information for use of the thirty-two sprites.

Bit-7 must be set to 0.

$$\begin{aligned} \text{DESIRED LOCATION}/128 &= \text{VALUE REG 5} \\ \text{LOCATION} &= \text{CONTENTS REG 5} * 128 \end{aligned}$$

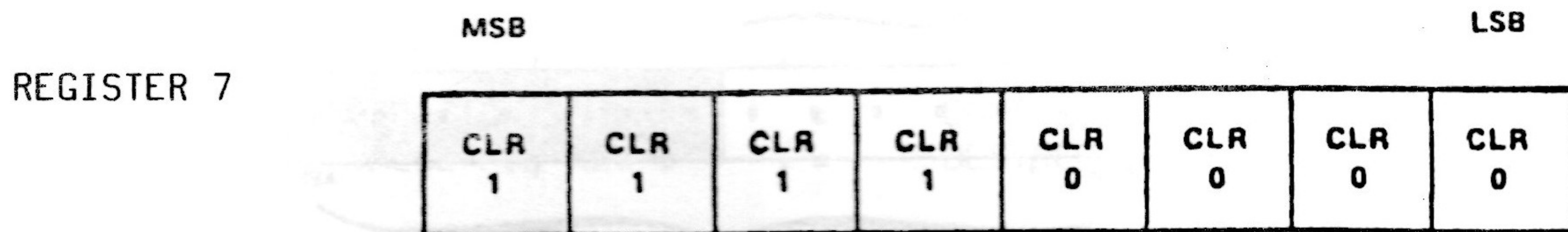
REGISTER 6



This register defines the start address of the SPRITE PATTERN GENERATOR. This table holds a library of sprite patterns which can be displayed by the pattern pointer in the SPRITE ATTRIBUTE TABLE. Changing a sprite pattern is a simple matter of over writing one byte in VRAM.

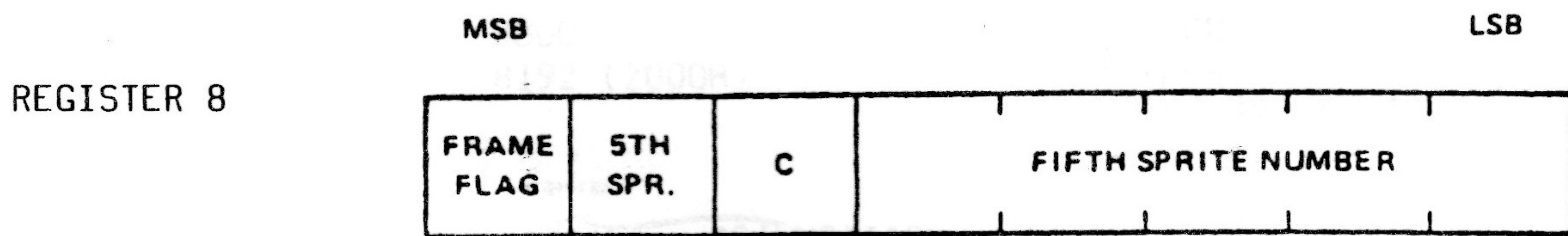
The table is a maximum of 2048 bytes in length and is divided into 256 blocks of eight bytes. It is a 3-bit register and therefore has a range from 0 - 7. This table must be located on a 2K boundary.

$$\text{LOCATION} = \text{CONTENTS REG 6} * 800H$$



The lower four bits of this register define the colour code of the paper when using the TEXT MODE and the border colour in all modes.

The upper four bits define the colour of the ink for characters in the TEXT MODE.



THIS IS A READ ONLY REGISTER

BIT 7 is the interrupt flag and is internally set to 1 at the end of a raster scan of the last line of the screen. It is reset after Register 8 has been read.

BIT 6 will be set if there are more than four sprites on a line and the number of the offending sprite will be placed in bits 4 - 0. This is a major disaster with the VDP: whenever five or more sprites appear on a scan line unpredictable displays are triggered and you can try it for yourself by setting six sprites on a horizontal line.

BIT 5 is set to 1 whenever the pixels of two or more sprites overlap.

WRITING TO THE VDP REGISTERS

Each of the VDP write only registers can be loaded using two eight bit data transfers. The first byte transferred is the data byte and the second is the control byte.

The data byte can be in the range 0 to 255 (#FF)

The VDP recognises a request to write to the VDP registers when the control byte has bit 7 set to 1. The next four lower bits must be Zero and bits 0 - 2 control the register number (0 - 7).

FORMAT

- 1] SEND DATA BYTE
- 2] SEND VDP REGISTER NUMBER

The easiest way to make sure bit 7 is always one is to perform a LOGICAL OR with 128 (#80).

EXAMPLE

INITIALISE VDP REGISTER 2 WITH A VALUE OF 12 (#C)

- 1] SEND DATA BYTE
LD A,#C ;data
OUT (09),A ;send it
- 2] SEND REGISTER NUMBER
LD A,2 ; REG number
OR #80 ; BIT 7 now set
OUT (09),A ; send it

It is important to note that the data and control bytes are directed to the VDP via Port 2.

READING AND WRITING TO VRAM

The VDP is coupled to Vram via an AUTO-INCREMENTING ADDRESS REGISTER. That is to say, once the address we want to write to or read from has been sent to the VDP we can read or write data to successive bytes using ONE BYTE transfers. The address will be automatically incremented. Thus, reading or writing to sequential addresses can be performed very quickly.

WRITE TO VRAM OR READ FROM VRAM

FORMAT

- 1] SEND LSBYTE OF ADDRESS IN VRAM
- 2] SEND MSBYTE OF ADDRESS IN VRAM
- 3] WRITE/READ DATA TO/FROM VRAM
- 4] WRITE/READ DATA TO/FROM VRAM sequential write/read
- 5] WRITE/READ DATA TO/FROM VRAM and so on

WRITING TO VRAM

The VDP recognises a write to Vram request when BIT 7 is 0 and BIT 6 is 1 in the MS Byte of the address. This is easily accomplished by performing a LOGICAL OR with 64 (40HEX)

EXAMPLE

SEND ASCII "A" TO VRAM ADDRESS #3C00

- 1] SEND LSByte of Address

LD A,00
 OUT (09),A
- 2] SEND MSByte of Address

LD A,#3C
 OR #40
 OUT (09),A
- 3] SEND DATA

LD A,"A"
 OUT (08),A

READING FROM VRAM

This time, the VDP recognises a READ request when BIT 7 = 0 and BIT 6 = 0. In most cases you do not need to take special precautions as Vram addresses are never greater than 14 bits. However, it is always wise to make sure and a LOGICAL AND with 63 (#3F).

EXAMPLE

READ BYTE AT VRAM LOCATION #3C00

SEND LSByte Vram Address

```
LD A,00
OUT (09),A
```

SEND MSByte Vram Address

```
LD A,#3C
AND #3F
OUT (09),A
```

READ from Vram Address

```
IN A,(08)
```

READING THE VDP STATUS REGISTER

This is a very simple operation

```
IN A,(09)
```

Once this instruction has been executed the Z80 A register will hold the contents of the STATUS REGISTER and data can be extracted by performing bit tests.

GENERAL PURPOSE SUBROUTINES FOR USING VDP

SET UP VDP REGISTERS FOR GRAPHIC 2 SCREEN

SET UP:

```

LD    HL,REG      ;Point HL at data
LD    BC,#0880    ;B=Number of Registers
                ;C=Register No + #80
LP:   LD    A,(HL) ;Get data value
      OUT   (09),A ;Send it to Vdp
      LD    A,C    ;Get Register No 0,1,2,etc
      OUT   (09),A ;already added to #80
      INC   C      ;increment register number
      INC   HL     ;increment HL to next data byte
      DJNZ  LP     ;Do eight times
      RET          ;All done return to caller

```

The above routine assumes that the register data is held in memory location REG. For Graphic Mode 2 the data at REG will be as follows:-

REG: DB 2,194,15,255,3,126,07,92

WRITE TO VRAM

Before calling this routine the HL registers must be loaded with the Vram address and A holds data byte to send.

WVRAM:

```

PUSH  AF          ; Save data byte
LD    A,L         ; Low byte of Vram address
OUT    (09),A     ; Send it
LD    A,H         ; High byte of Vram address
OR     #40        ; Make sure bit 7=0 and bit 6=1
OUT    (09),A     ; Send it
POP    AF         ; Get data byte back
VRAM: OUT    (08),A ; Send it to Vram address
      RET          ; Return to caller

```

The label VRAM is there so that sequential data transfers can be made without setting the address each time.

```

LD    A,#30      ; First data Byte
LD    HL,#3C00   ; Vram address
CALL  WVRAM      ; Go send
LD    B,6        ; Loop counter

```



```
LOOP: INC    A           ; A=#31 ,#32, etc
      CALL   VRAM        ; Send it
      DJNZ   LOOP        ; Do it 6 times
```

READ FROM VRAM

Again, HL register pair must contain Vram address and the A register will return with data from Vram.

```
RVRAM: LD     A,L         ; Low byte Vram address
      OUT    (09),A       ; Send it
      LD     A,H         ; High byte Vram address
      AND    #3F          ; Make sure bit 7 and 6 = 0
      OUT    (09),A       ; Now safe to send
RDRAM: IN     A,(08)      ; Get data from Vram in to A Reg
      RET                ; Take it and show to caller
```

Label RDRAM is for successive sequential reads from Vram.

CHAPTER TEN

THE DISPLAY MODES

TEXT MODE

This is the same as the normal EINSTEIN Basic text mode.

The VDP is initialised to text mode when the mode bits $M1 = 1 : M2 = 0 : M3 = 0$

(See previous chapter - VDP REGISTERS)

Text mode provides the following features:-

SCREEN

24 rows of 40 columns (960 character positions).

Up to 256 unique characters can be defined at any one time. The pixel size of text characters should be six wide by eight deep. These character patterns can be dynamically changed by transferring patterns from character libraries held in unused portions of Vram or Z80 ram.

Two colours are available: one for text colour and one for the backdrop. The colours can be chosen from a palette of fifteen hues including transparent.

EINSTEIN Basic uses the following set-up for text mode:-

FUNCTION	VRAM START ADDRESS	VRAM END ADDRESS
TEXT PATTERN	6400 (#1900)	8192 (#2000)
SCREEN	14336 (#3800)	15104 (#3B00)

Because, in Basic, the text mode has been designed as an integral part of the GRAPHIC MODE 2 tabling the pattern generator table is only 1K in length. This allows a maximum of 128 patterns to be defined. This is a compromise situation in order to have both modes resident in Vram at the same time.

Sprites are not allowed in text mode but it does have many advantages over other modes when text orientated programs are written - it is very compact and only occupies 3072 bytes (3K) of Vram. As already stated, due to its compactness, several pattern libraries can be stored in Vram. Several screens can also be stored in Vram and a new set up can be switched in simply by changing two VDP registers.

Animated displays can be triggered by having one set of patterns and storing fourteen text screens. By changing only VDP REGISTER 2 screens can be flipped in eleven micro-seconds!

Normally, the text pattern generator table is 2048 bytes (2K) in length divided into 256 patterns. Each pattern occupies 8 bytes. Each text position, on the screen, is six pixels wide and the lower two bits (0 & 1) of each text pattern are ignored by the VDP.

The °ON° bits take up the foreground colour and the °OFF° bits take up the backdrop colour.

Any pattern can be displayed in any position on the screen by loading its pattern number into the appropriate position in the screen table.

If we used the Basic set up as described above the following routine is an example of how easy it is to write characters or strings to a text screen.

SUBROUTINE EXAMPLE OF SENDING TEXT STRING

VDP SHOULD BE INITIALISED TO TEXT MODE


```

STRT:  LD DE, STRING      ;DE points to string data
        LD HL, #1C00      ;HL points to first position on screen
        CALL SEND         ;GO send string
        RET              ;return to caller
SEND:  LD A, (DE)         ;get first character of string
        CALL WVRAM        ;send address and first text byte
SND:   INC DE             ;make DE point to next character
        LD A, (DE)        ;put character in A register
        CP #FF            ;is it end of string?
        RET Z             ;if yes return
        CALL VRAM         ;otherwise send it
        JR SND            ;do it again
STRING: DB 'This is an Ascii print test', #FF

```

GRAPHIC MODE 2

This is the most useful of all modes and it is the mode EINSTEIN Basic uses for all of the graphic displays. Graphic Mode 2 is very versatile but versatility has a price - it is difficult to master. However, once mastered it is very easy to use and manipulation of animated displays is a simple matter.

The VDP is initialised to Graphics Mode 2 when mode bits M1=0 : M2=0 : M3=1

Vram is organised as shown below:-

VRAM SETUP

0000	-----	GRAPHIC GENERATOR MODE 2
6144	-----	GRAPHIC GENERATOR TEXT
7168	-----	TEXT SCREEN
8128	-----	UNUSED
8192	-----	COLOUR TABLE MODE 2
14336	-----	SPRITE GENERATOR
15360	-----	SCREEN MODE 2
16128	-----	ATTRIBUTE TABLE
16256	-----	UNUSED
16384	-----	

FUNCTION	VRAM START ADDRESS	VRAM END ADDRESS
TEXT PATTERN	6400 (#1900)	8192 (#2000)
SCREEN	14336 (#3800)	15104 (#3B00)

Because, in Basic, the text mode has been designed as an integral part of the GRAPHIC MODE 2 tabling the pattern generator table is only 1K in length. This allows a maximum of 128 patterns to be defined. This is a compromise situation in order to have both modes resident in Vram at the same time.

Sprites are not allowed in text mode but it does have many advantages over other modes when text orientated programs are written - it is very compact and only occupies 3072 bytes (3K) of Vram. As already stated, due to its compactness, several pattern libraries can be stored in Vram. Several screens can also be stored in Vram and a new set up can be switched in simply by changing two VDP registers.

Animated displays can be triggered by having one set of patterns and storing fourteen text screens. By changing only VDP REGISTER 2 screens can be flipped in eleven micro-seconds!

Normally, the text pattern generator table is 2048 bytes (2K) in length divided into 256 patterns. Each pattern occupies 8 bytes. Each text position, on the screen, is six pixels wide and the lower two bits (0 & 1) of each text pattern are ignored by the VDP.

The 'ON' bits take up the foreground colour and the 'OFF' bits take up the backdrop colour.

Any pattern can be displayed in any position on the screen by loading its pattern number into the appropriate position in the screen table.

If we used the Basic set up as described above the following routine is an example of how easy it is to write characters or strings to a text screen.

SUBROUTINE EXAMPLE OF SENDING TEXT STRING

VDP SHOULD BE INITIALISED TO TEXT MODE

CHAPTER TEN

THE DISPLAY MODES

TEXT MODE

This is the same as the normal EINSTEIN Basic text mode.

The VDP is initialised to text mode when the mode bits M1 = 1 : M2 = 0 : M3 = 0

(See previous chapter - VDP REGISTERS)

Text mode provides the following features:-

SCREEN

24 rows of 40 columns (960 character positions).

Up to 256 unique characters can be defined at any one time. The pixel size of text characters should be six wide by eight deep. These character patterns can be dynamically changed by transferring patterns from character libraries held in unused portions of Vram or Z80 ram.

Two colours are available: one for text colour and one for the backdrop. The colours can be chosen from a palette of fifteen hues including transparent.

EINSTEIN Basic uses the following set-up for text mode:-

FUNCTION	VRAM START ADDRESS	VRAM END ADDRESS
TEXT PATTERN	6400 (#1900)	8192 (#2000)
SCREEN	14336 (#3800)	15104 (#3B00)

Because, in Basic, the text mode has been designed as an integral part of the GRAPHIC MODE 2 tabling the pattern generator table is only 1K in length. This allows a maximum of 128 patterns to be defined. This is a compromise situation in order to have both modes resident in Vram at the same time.

Sprites are not allowed in text mode but it does have many advantages over other modes when text orientated programs are written - it is very compact and only occupies 3072 bytes (3K) of Vram. As already stated, due to its compactness, several pattern libraries can be stored in Vram. Several screens can also be stored in Vram and a new set up can be switched in simply by changing two VDP registers.

Animated displays can be triggered by having one set of patterns and storing fourteen text screens. By changing only VDP REGISTER 2 screens can be flipped in eleven micro-seconds!

Normally, the text pattern generator table is 2048 bytes (2K) in length divided into 256 patterns. Each pattern occupies 8 bytes. Each text position, on the screen, is six pixels wide and the lower two bits (0 & 1) of each text pattern are ignored by the VDP.

The °ON° bits take up the foreground colour and the °OFF° bits take up the backdrop colour.

Any pattern can be displayed in any position on the screen by loading its pattern number into the appropriate position in the screen table.

If we used the Basic set up as described above the following routine is an example of how easy it is to write characters or strings to a text screen.

SUBROUTINE EXAMPLE OF SENDING TEXT STRING

VDP SHOULD BE INITIALISED TO TEXT MODE


```

STRT:  LD DE, STRING      ;DE points to string data
        LD HL, #1000      ;HL points to first position on screen
        CALL SEND         ;GO send string
        RET              ;return to caller
SEND:  LD A, (DE)         ;get first character of string
        CALL WVRAM        ;send address and first text byte
SND:   INC DE             ;make DE point to next character
        LD A, (DE)        ;put character in A register
        CP #FF            ;is it end of string?
        RET Z            ;if yes return
        CALL VRAM        ;otherwise send it
        JR SND           ;do it again
STRING: DB 'This is an Ascii print test', #FF

```

GRAPHIC MODE 2

This is the most useful of all modes and it is the mode EINSTEIN Basic uses for all of the graphic displays. Graphic Mode 2 is very versatile but versatility has a price - it is difficult to master. However, once mastered it is very easy to use and manipulation of animated displays is a simple matter.

The VDP is initialised to Graphics Mode 2 when mode bits M1=0 : M2=0 : M3=1

Vram is organised as shown below:-

VRAM SETUP

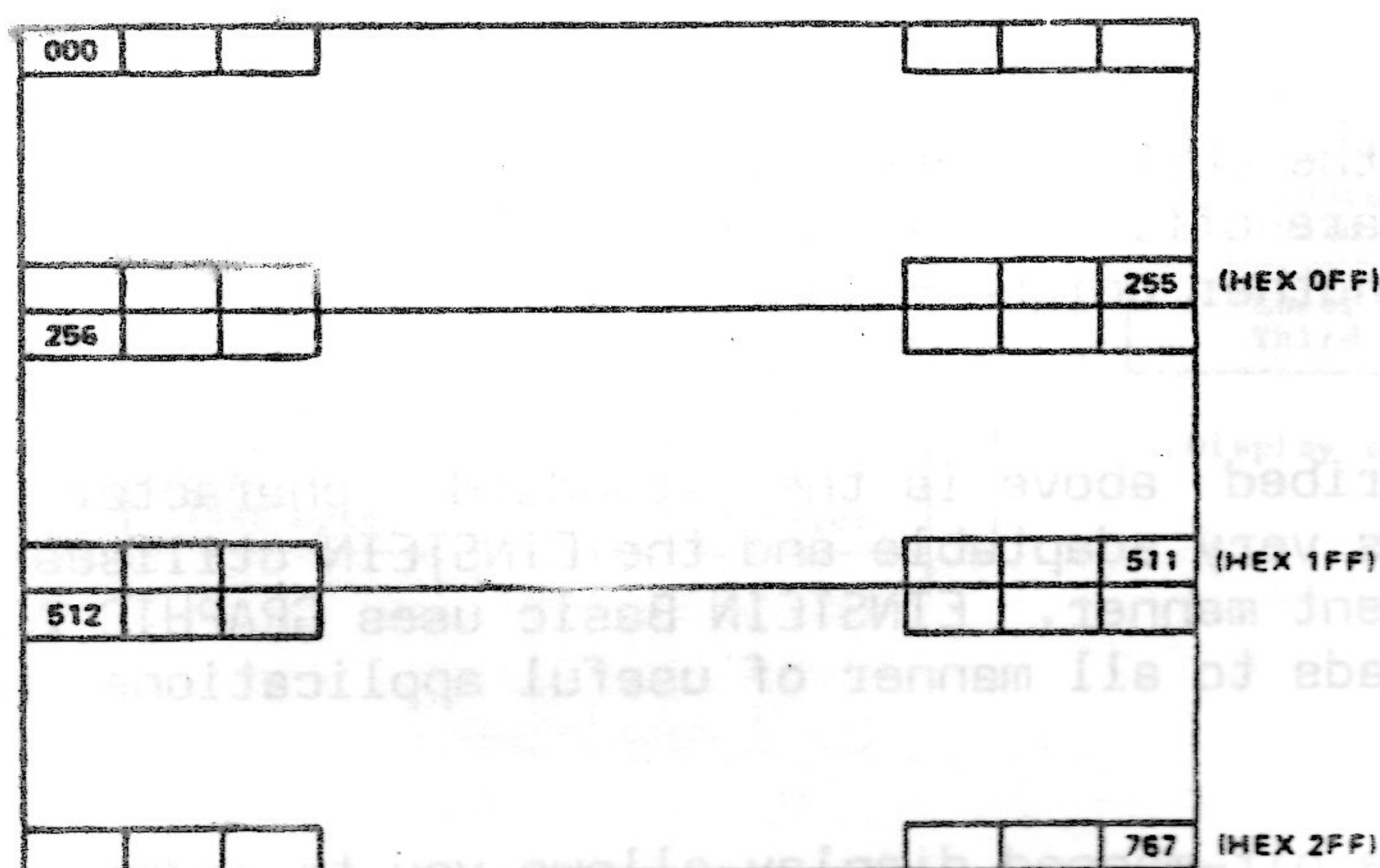
0000	-----
	GRAPHIC GENERATOR MODE 2
6144	-----
	GRAPHIC GENERATOR TEXT
7168	-----
	TEXT SCREEN
8128	-----
	UNUSED
8192	-----
	COLOUR TABLE MODE 2
14336	-----
	SPRITE GENERATOR
15360	-----
	SCREEN MODE 2
16128	-----
	ATTRIBUTE TABLE
16256	-----
	UNUSED
16384	-----

The NAME TABLE or SCREEN MEMORY is 768 bytes in length but the COLOUR & GENERATOR TABLES are each 6144 bytes long. The reason for this is not obvious and a little explanation may be in order. In the TEXT or GRAPHIC 1 modes, as with most other computers, only 256 Ascii characters are available. The EINSTEIN, in MODE 2, under assembly conditions allows you to treble this figure! In fact, you can create 768 unique patterns - one to each screen location. This mode also allows the patterns to be further enhanced by permitting eight bytes of colour information to be used within each pattern - all sixteen colours can be defined within one character.

NAME TABLE (SCREEN)

The Name Table contains 768 entries which correspond to each of the 768 display positions on the VDU. Because there are only 256 Ascii codes (Pattern Names) in order that we can display 768 different patterns the screen is split into three sections, and each section is 256 bytes in length.

INSERT MODE 2 SEGMENTED TABLE



PATTERN TABLE (PATTERN GENERATOR)

This table is 6144 bytes long and is also split into three equal blocks of 2048 bytes. Each section is further divided into 256 (8 x 8 pixel) graphic blocks. The first 256 patterns can only be displayed in the upper third of the screen, the second block is confined to the middle section of the screen and the final 256 patterns can only be displayed in the bottom third of the screen. Because of these limitations, care must be taken when moving a character around the screen: If Ascii pattern 128 defines a space invader in the top third of the screen, to move the invader into the middle or bottom third it must be defined as Ascii 128 [CHR\$(128)] in each of these sections.

COLOUR TABLE

The COLOUR TABLE is exactly the same length as the PATTERN TABLE (6144 bytes). This too, is segmented into three blocks of 2048 bytes. Each section is further divided into 256 colour definitions which are eight bytes long. Each section of the Colour Table maps onto the equivalent section of the PATTERN TABLE.

Each byte within the COLOUR TABLE defines the colour of the bits that are on or the bits that are off. This colour can be the same as the background, transparent, or another unique colour.

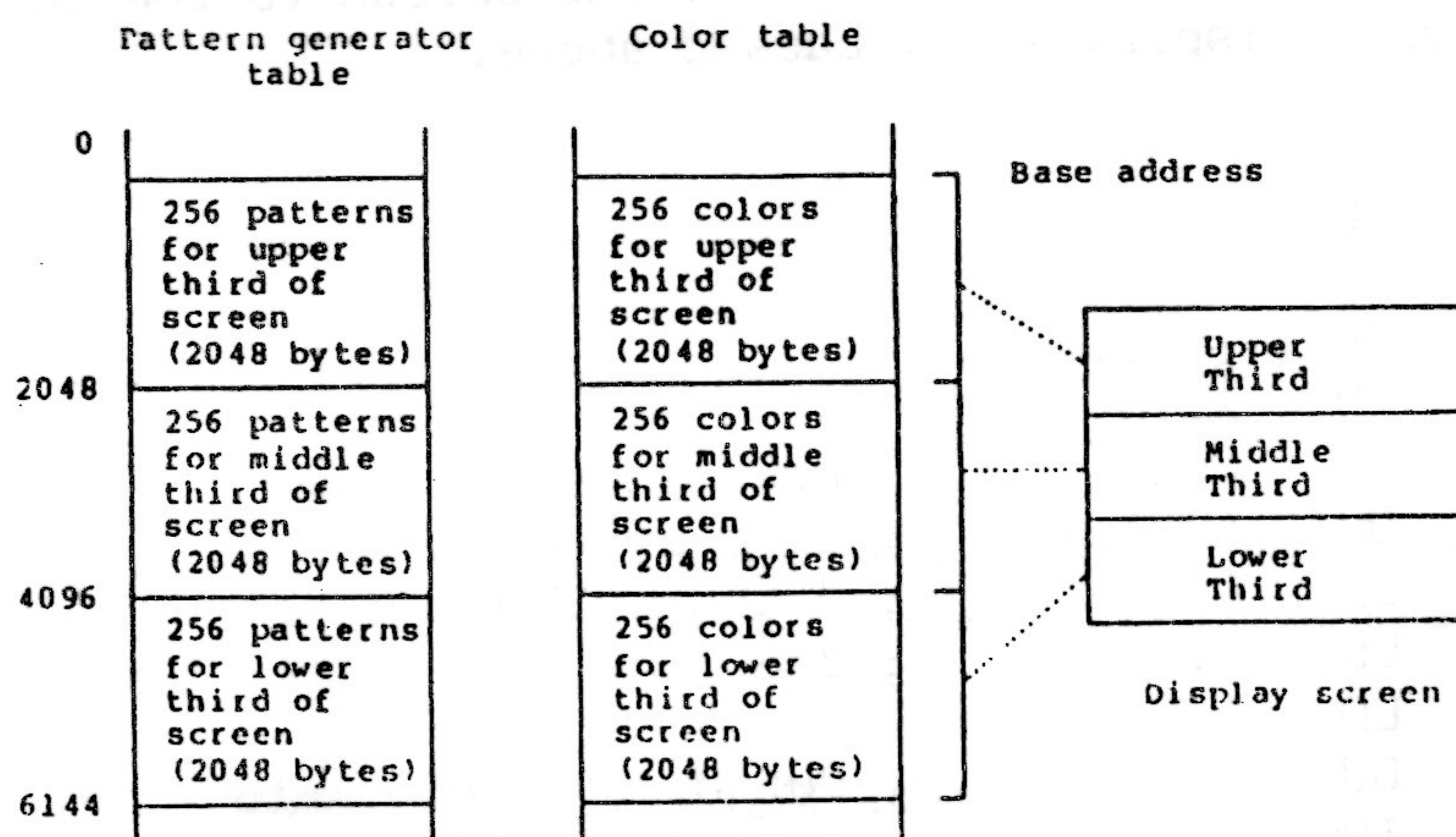
The method described above is the standard character mapped display. GRAPHIC MODE 2 is very adaptable and the EINSTEIN utilises this mode in a completely different manner. EINSTEIN Basic uses GRAPHIC 2 as a bit-mapped display which leads to all manner of useful applications, not least games programming.

Using Mode 2 as a bit-mapped display allows you to address every pixel on the screen individually. This is useful for plotting points, drawing lines and creating complex back drop designs. Unfortunately, there is a drawback when using the VDP in this manner - although you can address individual pixels, the colour bits cannot be formatted in the same way. We can, however, get around this by using only two colours, one for 'ON' pixels and one colour for 'OFF' pixels or use more than two colours but be very careful where we plot them.

To set up Mode 2 as a bit-mapped display we must write a different value to each of the screen locations (Pattern Name table). This means that the Pattern Name table holds location values instead of containing the actual pattern numbers (Ascii values) which is the normal method of writing to the screen. By using this technique a unique numbered pattern is created, on screen, whenever eight bytes of information are placed in the Generator table.

Each byte of the Colour Table provides a foreground and background colour for the corresponding byte in the Pattern Generator table. Byte 0 of the colour table maps directly onto Byte 0 of the Generator table. Byte 1 maps to Byte 1 and so on.

If we assume that the Pattern Generator is located at Vram address 0 and the Colour table is at 8K, it is a simple matter to discover which Generator byte is coloured in by which colour byte. All we need do is add an offset, which is equal to the distance between the two tables, to find the desired Vram address. In the case of our mapping scheme the offset is #2000 (8192).



VRAM MAPPING SCHEME

GRAPHIC MODE 2

In order that the first pattern character block will map onto the first character cell of the screen and the second pattern will map onto the second character cell, screen position 0 must contain 0 and screen position 1 must contain a 1. This mapping is continued up to screen position 256 which will hold the value 255. This, of course, only accounts for the top third of the screen.

We have already discussed how the VDP internally segments the Pattern Generator, Colour table, and Screen into three equal blocks of 2048, 2048 and 256 bytes respectively. Character names in the upper third of the screen automatically correspond to the character patterns in the upper third of the Pattern Generator, character names in the middle segment of the screen correspond to the character patterns in the middle third of the Generator table and screen names in the bottom third of the screen map to character patterns in the lower third of the Generator table.

In order to fill these remaining segments of the screen with the correct Pattern names we load screen position 256 with 0 through to 255 and screen position 512 with 0 through to 255. This means that we have the screen divided into three sections of 256 bytes. These bytes are numbered 0 to 255 in each section.

The following routine will initialise the screen to the correct parameters for a bit-mapped display as discussed above.

```
G2SETUP:    LD    HL,#3C00    ; start of screen
            LD    A, L
            OUT   (09),A
            LD    A, H
            OR    40H        ; make sure vdp knows
            OUT   (09),A    ; is a write operation
            LD    C,03      ; 3 SEGMENTS OF SCREEN
LOOP_1      LD    B,0       ; 256 locations (INNER LOOP)
            LD    A,0
LOOP_2      OUT   (08),A    ; Put 0,1,2,3 etc into
            INC   A         ; screen locations
            DJNZ  LOOP_2    ; have we filled 1 segment?
            DEC   C         ; Do 3 segments until
            JR    NZ,LOOP_2 ; C = 0
            RET
```

The program relies on the fact that the VDP auto-increments the address without further prompting so once the address #3C00 (15360) has been sent each time data is posted the address counter is incremented. The inner loop [LD B,0] takes care of the 256 locations. The outer loop [LD C,03] makes sure that each third of the screen is set up in an identical manner.

You may have found the latter explanation hard to grasp and , unfortunately, there is no easy way of describing the technicalities. If you are asking why we set up the tables in this way the following may help to ease your frustration. If not, you will have to be patient until we try some examples and then you must analyse them carefully until you are sure you have grasped the fundamentals.

Normally we would set up patterns in the Generator table and then place them on the screen by putting the character code into the Pattern Name table (screen). If we wished to print the word TEST we would place the Ascii codes [CHR\$(n)] representing the letters TEST in the screen locations we want the message to be displayed. This is no longer the situation for a bit-mapped screen. We have already placed pattern numbers into the screen positions. If we now fill the first eight bytes in the Generator table with the information that goes to make the letter 'A' it will take on the character code 0. If we now write the same values to the last eight bytes in the top third of the Generator table the letter 'A' will take on the character code 255. At this point we have two letter "A's" written to the screen but instead of both taking on the character value #41 (normal Ascii value) they have unique screen cell values of 0 and 255. If the first 'A' was altered by adding an underline, it would not affect the 'A' at position 255. This is why, under a bit-mapped screen, you can plot or unplot points in an exclusive manner.

Before anything can be displayed on the screen the colour bytes have to be set. On initialisation you can set all the Colour table to #4F which will give you the standard blue and white screen used by the EINSTEIN at power-up. O.K! You don't like blue and white so put in your own values but don't forget that you are now filling 6144 bytes (2048 bytes for each third of the colour table).

To turn on the very first pixel in Top left hand corner of the screen we would write #80 (128) in the first byte of the Pattern Generator.

#80 = 1000 0000 binary

In a similar manner, to turn on the last pixel in the bottom left hand corner of the screen we would write 01 to location #17FF in the Pattern Generator.

#01 = 0000 0001 binary

To make matters a little easier we could write a subroutine that will calculate the address for any given x,y pixel co-ordinate.

Using the formula:-

x = 0 TO #FF (0 - 255)
y = 0 TO #C0 (0 - 191)

Horizontal offset = INT (x/8)*8 ; STORE REMAINDER
Vertical offset = INT (y/8)*256 ; STORE REMAINDER

BYTE ADDRESS = HORIZONTAL OFFSET + VERTICAL OFFSET + REMAINDER Y/8

This now gives you the actual Vram location in the Generator table. To find out what data has to be written to set that pixel simply take the remainder of x/8 and use the following lookup table.

REMAINDER x/8	DATA TO WRITE
0	#80 (128)
1	#40 (64)
2	#20 (32)
3	#10 (16)
4	# 8
5	# 4
6	# 2
7	# 1

I am sure that some of you bright sparks have spotted a sequence which can be put to good use when plotting pixels.

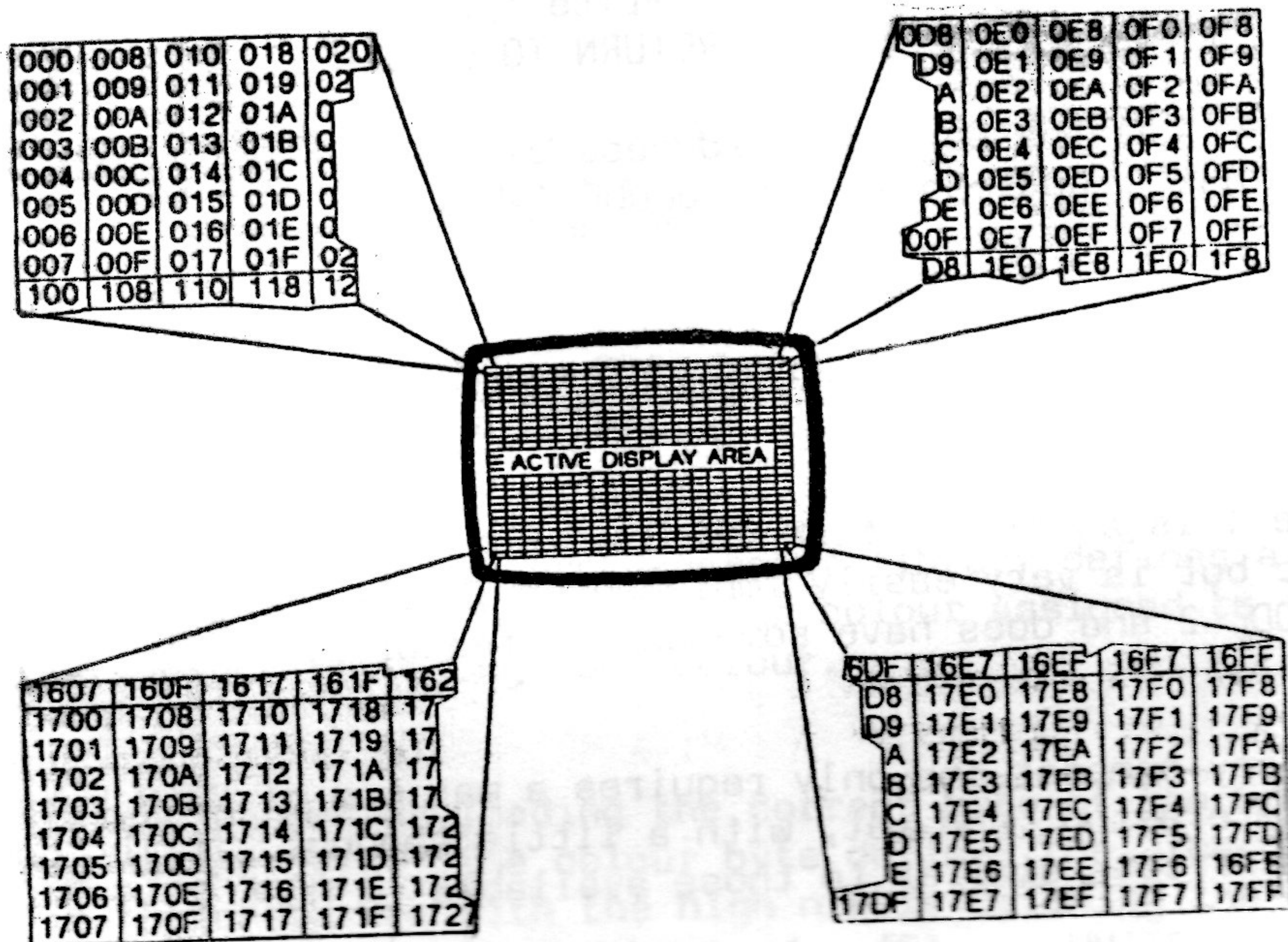
EXAMPLE:

PLOT A PIXEL AT X CO-ORDINATE 33
Y CO-ORDINATE 73

X OFFSET = (33/8)*8 = 32 REMAINDER 1
Y OFFSET = (73/8)*256 = 2304 REMAINDER 1

BYTE LOCATION = 32 + 2304 + 1 = 2337
DATA TO WRITE = 64 (Using lookup table)

We would therefore send 64 to Vram location 2337 to plot a point at



ROUTINE TO CLEAR SCREEN IN BIT MAPPED MODE

CLS:

```
LD BC, 6144 ; length of GENERATOR TABLE
LD HL, 0 ; location GENERATOR TABLE
LD E, 0 ; write Zeros to all locations
PUSH BC ; save original count
CALL CLSX
POP BC ; get count back
SET 5, H ; align to Colour table
CALL CLSX ; from Generator table
RET
```

CLSX:

CLSY:

```
CALL ADDOUT ; SEND ADDRESS
LD A, E ; GET DATA BYTE
OUT (08), A ; SEND IT TO VDP
DEC BC ; DEC COUNTER
LD A, B ; TEST
OR C ; FOR ZERO
JR NZ, CLSY ; OTHERWISE DO IT AGAIN
RET
```

ADDOUT:

```
LD A, L
OUT (09), A
LD A, H
```



```
OR    40H      ; Make sure VDP knows it's
OUT   (09),A   ; A write operation
RET                      ; RETURN TO CALLER
```

Setting bit 5 of the Generator address is an easy and quick method of aligning the correct address in the COLOUR TABLE.

GRAPHICS MODE 1

Graphics Mode 1 is an alternative display mode which is not allowed under EINSTEIN Basic but is very easily implemented in assembler. This mode is easier to use than MODE 2 and does have some useful features.

This mode is very compact and only requires a maximum of 2848 bytes in Vram for the complete setup, and yet, with a little thought, it is possible to create graphic effects similar to those available in MODE 2.

SCREEN

24 rows x 32 columns giving a total of 768 character positions on the display. 256 unique patterns can be displayed at any one time. These patterns can be washed in any two of the sixteen available colours. The Backdrop or Border can be one of the sixteen colours including transparent.

It is very easy, in this mode, to dynamically redefine the 256 patterns to give character variations limited only by the amount of available ram storage.

Sprites are available up to a maximum of 32.

Mode 1 is triggered by setting the mode bits as follows:-

M1 = 0 : M2 = 0 : M3 = 0

As with the text mode it is possible to create different pattern libraries in Vram and switch to a complete new Mode 1 setup simply by changing two VDP registers. If only one character set is in residence, it is possible to store 14 separate Mode 1 screens in Vram memory. These screens can be flipped to the display by modifying only VDP register 2. A screen can be flipped in 11 micro-seconds which is very useful for animated displays.

The amount of storage space required for a normal MODE 1 setup can be broken down in the following manner:-

Pattern Generator Table	2048 bytes.
Pattern Colour Table	32 bytes.
Pattern Name Table (screen)	768 bytes.

The Pattern Generator Table is 2048 bytes in length and is split into 256 graphic blocks each taking 8 bytes.

Each 8 byte block of the Pattern Generator library defines a graphic pattern in which the 1's take up the foreground colour assigned to it and the 0's take on the appropriate background colour.

The colours are chosen by loading the correct byte of the Colour Table with the desired colour byte. The colour byte defines the foreground/background colour for the character with the high nibble defining the foreground and the lower nibble defining the background. #F1 = WHITE ON BLACK.

PATTERN NUMBERS	COLOUR BYTE USED
0 - 7	0
8 - 15	1
16 - 23	2
24 - 31	3
32 - 39	4
40 - 47	5
ETC ...	

From the above table it can be seen that for each block of 8 bytes from the Pattern Generator library the colour for each byte is determined by only one byte from the Colour Table. Although this method is easy to understand, it does mean that to obtain a wide range of differently coloured graphic patterns requires more than a little thought.

Any one of the patterns held in the Generator library can be displayed in any position on the screen simply by loading the value of the pattern into the appropriate position in the screen area.

To try out Graphic Mode 1 let's write a routine to clear the screen and then print a border of asterisks around the display. To do this we must first design the characters. The graphic pattern we will use to clear the screen is a null character, that is to say, its pattern is made up of all zeros.

The second pattern defines an asterisk. The null character will be pattern zero and will use bytes 0 - 7 in the Generator Table. The asterisk will be pattern one and will use bytes 8 - 15 in the Pattern Generator.

In our setup the tables are located as follows:-

Pattern Generator	start #800	(2048)
Graphic Screen	start #1400	(5120)
Colour Table	start #2000	(8192)

To trigger this setup we must first initialise the VDP registers to the correct values which are:-

REG 0	= 0	; M3 = 0
REG 1	= #C0	; M1 & M2 = 0
REG 2	= 05	; Screen at #1400
REG 3	= #80	; Colour Table at #2000
REG 4	= 01	; Pattern Generator at #800
REG 5	= #20	; Sprite Attribute table
REG 6	= #00	; Sprite Pattern at 0
REG 7	= 01	; Black background/border

Pass 1 errors: 00

```

0100      1 ;Routine to clear screen in graphic model
0100      2 ;
0100      3      org      100h
0100      4
0100 CD1101 5      call    gl_init
0103 CD3B01 6      call    filpat
0106 CD2A01 7      call    col      ;fill colour for two characters
0109 CD6A01 8      call    cls
010C CD8601 9      call    set_border
010F      10 loop
010F 18FE 11      jr      loop
0111      12 gl_init
0111 212201 13      ld      hl,reg      ;get register values
0114 018008 14      ld      bc,0800h ;b= number of registers c=Reg No +80h bit 7 set

```



```

0117      15 set_loop
0117 7E      16      ld      a,(hl)      ;get data value
0118 D309      17      out      (9),a
011A 79      18      ld      a,c      ;register number
011B D309      19      out      (9),a
011D 0C      20      inc      c      ;increment to next register
011E 23      21      inc      hl      ;increment data pointer
011F 10F6      22      djnz     set_loop ;do it until all done
0121 C9      23      ret              ;return to caller
0122      24 reg
0122 00C00500 25      db      0,0c0h,05,80h,01,20h,00,0ah;data mode 1 setup
012A      26 col
012A 210020      27      ld      hl,2000h ;start of colour table
012D CDCB01      28      call     addout ;send it to vdp
0130 3E1C      29      ld      a,1ch ;black foreground/green paper
0132 D30B      30      out      (8),a ;send it for pattern 1
0134 07      31      rlca
0135 0F      32      rrca      ;hand about a bit for vdp
0136 3E16      33      ld      a,16h ;
0138 D30B      34      out      (8),a
013A C9      35      ret
013B      36 filpat
013B      37
013B      38 ;fill graphic generator with patterns
013B      39
013B 21000B      40      ld      hl,800h ;base address graphic gen
013E CDCB01      41      call     addout ;send address
0141 215A01      42      ld      hl,chars ;point hl at character data
0144 0E02      43      ld      c,2 ;outer loop counter
0146      44 patlp1
0146 060B      45      ld      b,8 ;16 bits of data for two patterns
0148      46 patlp
0148 7E      47      ld      a,(hl) ;get pattern data
0149 D30B      48      out      (8),a ;send data to vdp
014B 23      49      inc      hl ;point to next data byte
014C 10FA      50      djnz     patlp ;do it for all pattern data
014E 0D      51      dec      c
014F C8      52      ret      z ;return if all done otherwise
0150 E5      53      push     hl ;save pointer

0151 21400B      54      ld      hl,840h ;next 8 characters up in vram = chr$(8)
0154 CDCB01      55      call     addout
0157 E1      56      pop      hl ;restore pointer
0158 18EC      57      jr      patlp1 ;do it all again

015A      58 chars
015A 00000000      59      db      0,0,0,0,0,0,0,0;null character
0162 20F87070      60      db      20h,0f8h,70h,70h,0f8h,20h,0,0;* character
016A      61 cls
016A 210014      62      ld      hl,1400h ;start of screen
016D CDCB01      63      call     addout
0170 210003      64      ld      hl,768 ;counter for number of screen locations
0173 3E00      65      ld      a,0 ;null character = ascii 0
0175 C07901      66      call     sendit
0178 C9      67      ret

```



```

0179          68 sendit
0179 F5      69      push af      ;save character
017A D308    70      out (8),a    ;send it to screen
017C 2B      71      dec hl
017D 7C      72      ld a,h
017E 85      73      or l        ;check if hl = 0 see why we save A reg
017F 2803    74      jr z,bk
0181 F1      75      pop af      ;no not zero yet so get char and
0182 1BF5    76      jr sendit   ;do it all again
0184          77 bk
0184 F1      78      pop af
0185 C9      79      ret
0186          80 set_border
0186 CD9001   81      call top      ;fill top of screen
0189 CD9F01   82      call bot      ;now do bottom
018C CDAE01   83      call sides   ;and finally the sides
018F C9      84      ret
0190          85 top
0190 210014   86      ld hl,1400h    ;first screen location top of screen
0193 CDCB01   87      call addout
0196 3E08     88      ld a,8      ;character = *
0198 212000   89      ld hl,20h     ;32 columns
019B CD7901   90      call sendit
019E C9      91      ret
019F          92 bot
019F 21E016   93      ld hl,1600h    ;bottom of screen
01A2 CDCB01   94      call addout   ;send it to vdp
01A5 212000   95      ld hl,20h     ;32 columns
01A8 3E08     96      ld a,8      ;ascii = *
01AA CD7901   97      call sendit
01AD C9      98      ret
01AE          99 sides
01AE 212014  100      ld hl,1420h    ;second line of screen
01B1 0616     101      ld b,22      ;loop counter
01B3          102 sb
01B3 C5      103      push bc      ;save counter
01B4 CDCB01  104      call addout   ;send address
01B7 3E08    105      ld a,8      ;pattern number = *
01B9 D308    106      out (8),a
01BB 111F00  107      ld de,31     ;we are going to add 31 to hl
01BE 19      108      add hl,de    ;so that it now points to right hand side
01BF CDCB01  109      call addout   ;send this address
01C2 3E08    110      ld a,8      ;our astrisk again
01C4 D308    111      out (8),a   ;put it on screen
01C6 23      112      inc hl     ;hl now warps round and is now left hand
01C7          113      ;side one row down
01C7 C1      114      pop c      ;get loop counter]
01C8 10E9    115      djnz sb
01CA C9      116      ret
01CB          117 addout
01CB 7D      118      ld a,l      ;get lsb of address
01CC D309    119      out (9),a
01CE 7C      120      ld a,h      ;now msb
01CF F640    121      or 40h     ;make vdp knows its a write operation
01D1 D309    122      out (9),a
01D3 C9      123      ret      ;all done
Pass 2 errors: 00

```


CHAPTER ELEVEN

SPRITES

Sprites are very important in animated game displays and have all manner of uses in graphic displays.

A sprite is a special animation pattern which can be moved, one pixel at a time, in a horizontal, vertical or diagonal direction, and is motivated in a way that is totally independent of the background pattern.

The sprite can be coloured in any one of 15 colours plus transparent. Multicoloured sprites can be designed by overlaying two or more sprites in different colours. Care must be taken to ensure that the overlay pattern is limited to four sprites or the fifth sprite syndrome may have unpredicted results on the display.

Sprites can assume any one of several sizes and magnification. They can be 'bled' into the display from any direction.

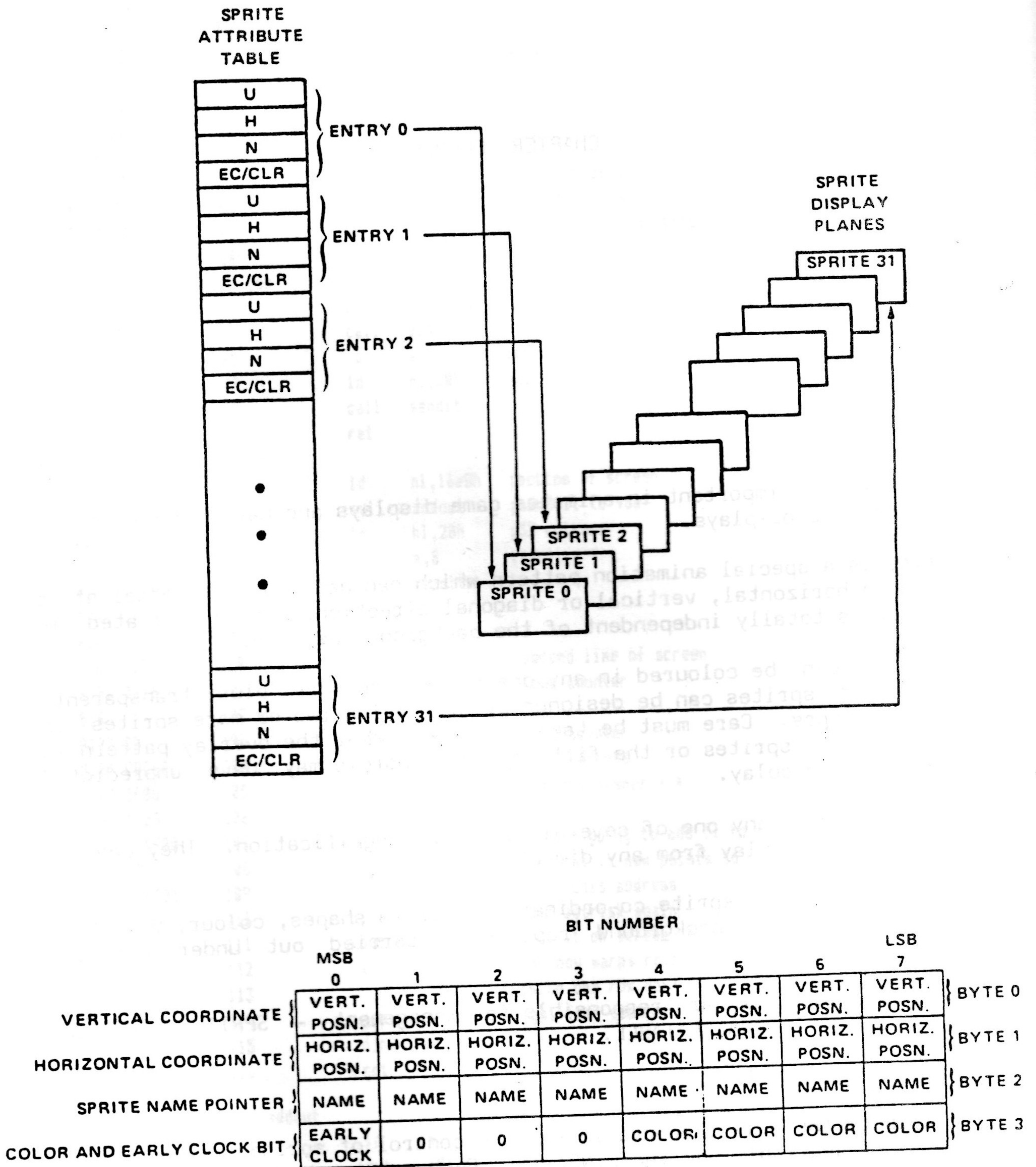
With the exception of sprite co-ordinates, pattern shapes, colour, all other maintenance such as background replace is carried out under hardware control.

Two sections of Vram are responsible for management - SPRITE ATTRIBUTE TABLE and SPRITE GENERATOR TABLE.

SPRITE ATTRIBUTE TABLE

The attribute table is responsible for the control of sprites. Thirty two sprites are available on the EINSTEIN. Each sprite has four bytes of dedicated information which means this table is 128 bytes long.

The start address of the table is determined by the contents of VDP Register 5 which locates the table on an 128 byte boundary.



Each sprite has an hardware priority index assigned to it by the VDP. Sprite 0 has a higher priority than Sprite 1 and the higher the sprite number the lower its priority. The sprite with the higher priority will appear to pass in front of the sprite with a lesser priority.

Sprite 0 is assigned attribute bytes 0,1,2,3 and has the highest priority. Sprite 31 has the lowest priority of all sprites and is assigned attribute bytes 124,125,126,127.

The first two bytes control the Y and X co-ordinates of the sprite on screen.

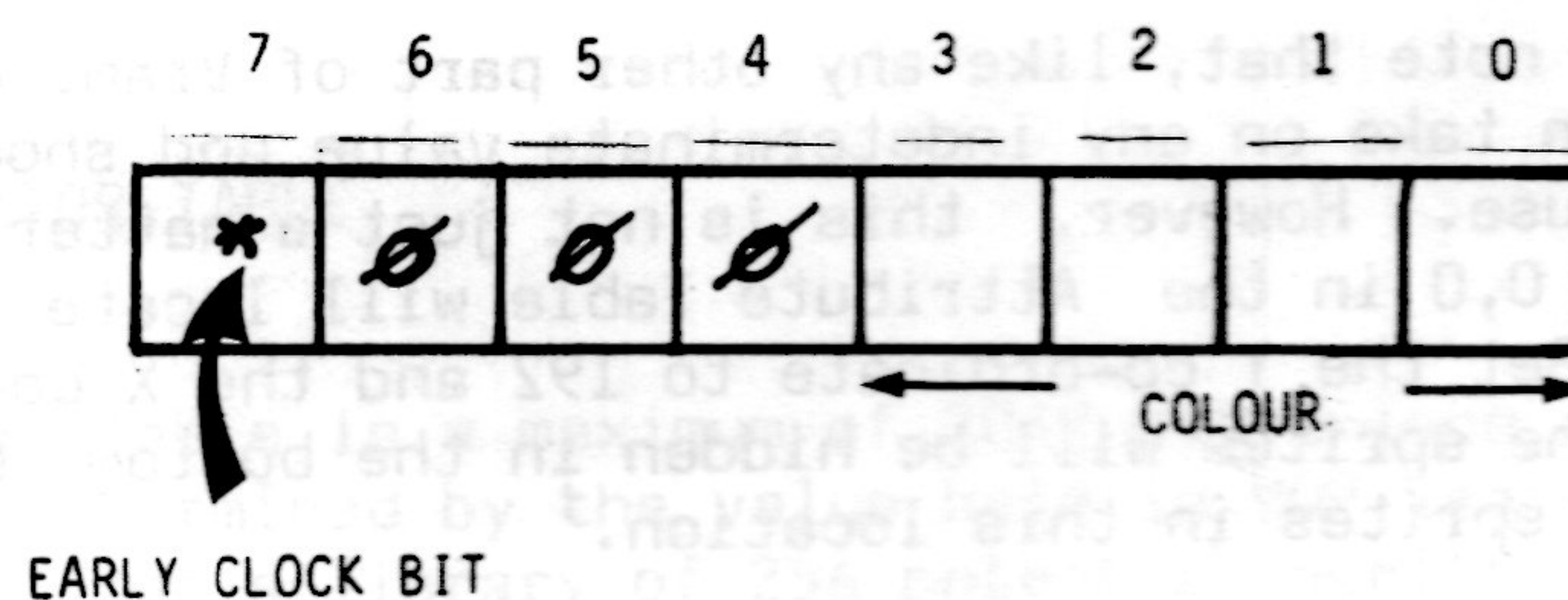
The first byte indicates the vertical distance (in pixels) from the top of the screen. The hardware is designed in such a manner that a value of #FF (-1) will butt the sprite against the border at the top of the screen. A value of 0 will place the sprite one pixel below the top of the screen.

The second byte in the attribute table decides the horizontal position (in pixels) from the left hand side of the display. This time, a value of 0 places the sprite against the left hand border.

All sprite positions are taken from the upper left corner of the sprite.

The third byte in the Attribute Table points to the sprite's pattern in the Sprite Generator Table.

The fourth byte determines the colour of the sprite but also carries out a further important function. The lower four bits of the colour byte determine the colour of the ON bits within the pattern. The OFF bits are automatically set to transparent.



Early Clock Bit

The Most Significant Bit of the colour byte is known as the EARLY CLOCK BIT. When this bit is zero it does nothing. However, when the bit is set it allows us to bleed a sprite off the screen or onto the screen. When the bit is active [1] the horizontal position of the sprite is shifted left by 32 pixels - even the largest, 32 x 32 bit sprite would be completely off the screen. When this bit is set values between 0 and 31 bleed the sprite out of the border onto the screen. The EARLY CLOCK BIT must be re-set before you can bleed the sprite off the right-hand side of the screen!

The advantage of these tables is the fact that they can be changed dynamically during the execution of a program and some can be changed from an interrupt servicing routine. Consider the advantages:-

- a) Create moving sprites by updating the x,y co-ordinates in bytes one and two.
- b) Animate any sprite by changing the pattern pointer in byte three.
- c) Change colour and create scintillating animations by changing byte four.

Because sprite information is held within fixed tables sprite impact routines are easy to implement - you can always refer to these tables and check x,y co-ordinates. If this routine is handled under interrupt control sprite co-incidence can be constantly checked without sacrificing program speed. We shall later experiment with a routine to give us sprite collision detection.

INITIALISING THE SPRITE ATTRIBUTE TABLE

We have already stated that the start address, in Vram, of the Sprite Attribute Table is governed by the contents of VDP Register 5.

It is important to note that, like any other part of Vram, on power-up the attribute table can take on any indeterminate value and should be properly initialised before use. However, this is not just a matter of setting each location to zero - 0,0 in the Attribute Table will locate all the sprites on screen at 0,0. Set the Y co-ordinate to 192 and the X co-ordinate to 0 - which means that the sprites will be hidden in the bottom border. Always place your inactive sprites in this location.

Servicing sprites takes up processor time and the more sprites that are active the longer it takes the VDP to carry out the process. If we only have three active sprites it is a complete waste of processor time to allow the VDP to check the status of all 32 sprites. We can help the VDP by "locking-off" all sprites not active. This is done by placing a value of 208 in the Y co-ordinate of the next sprite after the last sprite that is active. When the VDP comes across a value of 208 it knows that it doesn't

need to carry on and goes back to do something more useful.

Take a look at the following subroutine :-

```
;Lock off any nor active sprite
;On entry to this routine the A
;register should hold the number of
;active sprites. A = 0 means no
;active sprites .
```

```
;Can be used as a subroutine to be
;dynamically charge from within the
;main program.
```

LOCOFF:

```
CP 32 ;If A = 31 then no action
RET NC
```

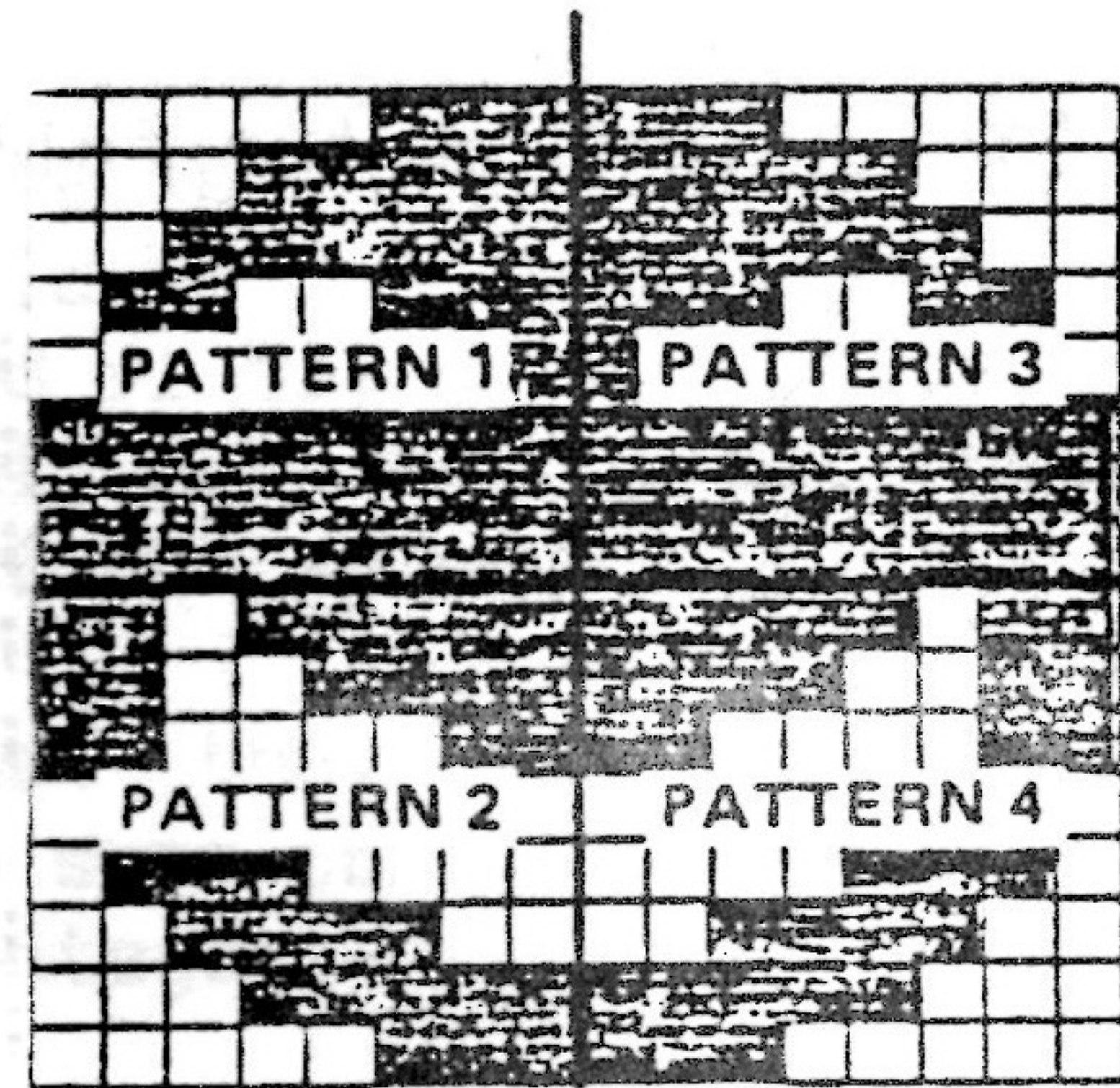
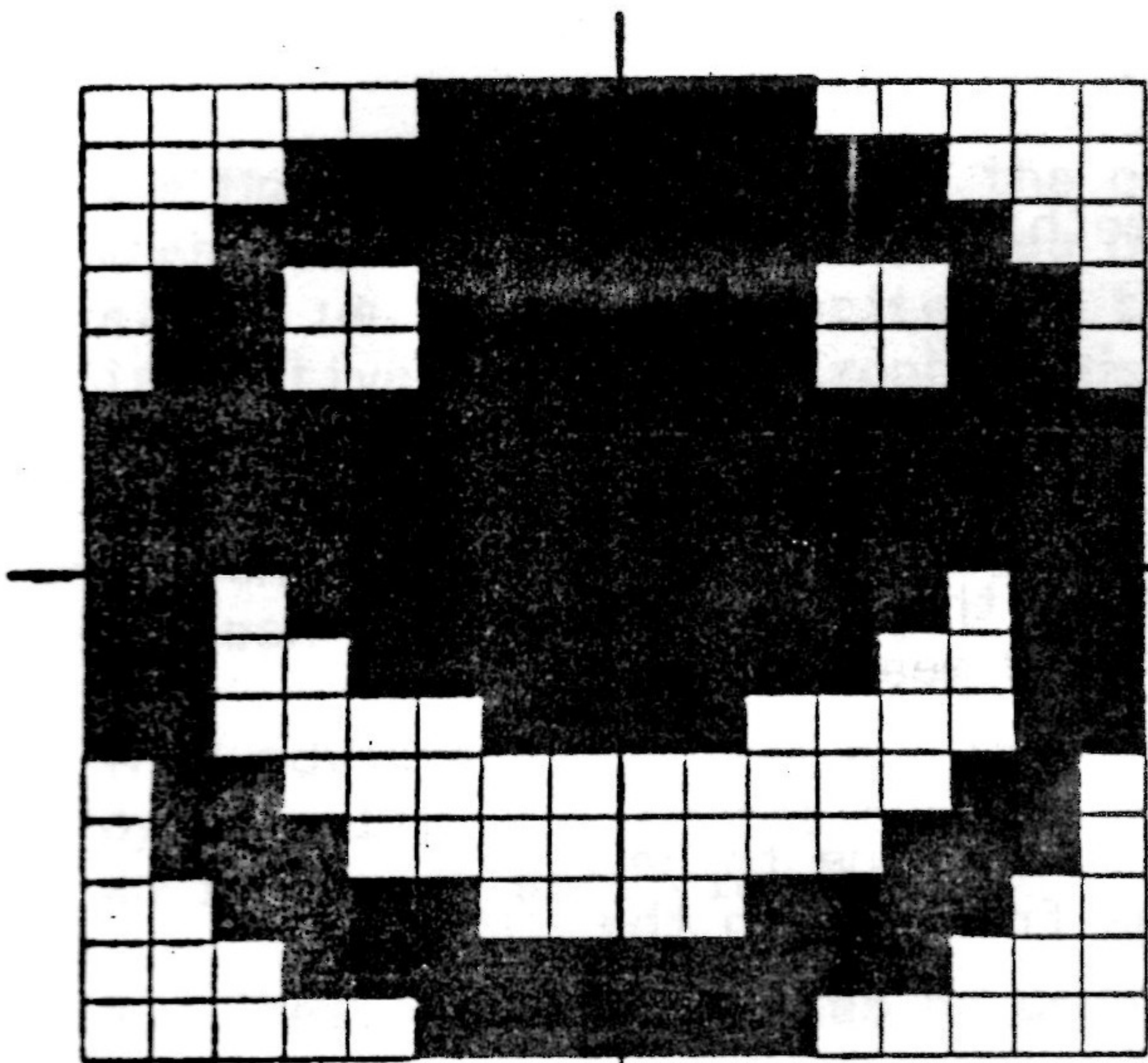
LOCK:

```
SLA A ;A * 2
SLA A ;A * 4
LD E,A ;Move A into E so that
LD D,0 ;we can add it to the start
ADD HL,DE ;address of the attribute
;table which is held in HL
CALL WVRAM ;so that we can send it to Vram
;Vram now points to correct
;address for sprite to be
;'locked-off'
LD A,208 ;Lock-off value
CALL VRAM ;SEND IT
RET ;All done so return
```

SPRITE GENERATOR TABLE

The Generator Table is a maximum of 2048 bytes long and always starts on a 2K boundary determined by the value held in VDP Register 6. The table is capable of holding a library of 256 potential sprite patterns each taking up 8 bytes. The patterns are utilised by placing their respective number in the Attribute Table - 0,1,2,255.

When 16*16 bit sprites are used each pattern will use 32 bytes which limits the library to 64 patterns. With this size of sprite you cannot access them sequentially as with 8*8 sprite patterns. This time you need to count in blocks of four - 0,4,8,248,252. Study the diagram below which shows how 16*16 bit sprite patterns in the Generator Table are mapped to the screen.



16*16 SPRITE

SPRITE ANIMATION

Animating sprites is simply a matter of changing one byte within the Sprite Attribute Table to loop through a sequence of patterns held in the Sprite Generator Table.

Correct timing is the key to realistic animation. This is not always as easy as it may seem - if the timing, when sequencing a series of patterns, is not correct, animation will look awkward. The most suitable solution is to perform this task under interrupts however, for the faint-hearted, an alternative solution is to use a timing loop and whenever the counter reaches zero a new pattern is displayed. Animation can then be fine-tuned by experimenting with the value in the loop counter.

The following program demonstrates this method. The sprite pattern numbers are held in a table. Our sequence is limited to four sequences. Whenever the fourth pattern has been displayed the program loops back to display the very first pattern and the sequence restarts. Animation can be speeded up or slowed down by altering the value in the DELAY routine. Further modifications can be made by altering the length of the sequence and changing the contents of ANITABLE to 0,1,2,3 2,3,2,3,1. This should give a pseudo oscillating animation sequence.


```

0100      1 ;ANIMATE A SPRITE IN A SEQUENCE OF FOUR PATTERNS
0100      2 ;
0100      3      org      100h
0100 CD1001 4      call    g2init    ;set graphic mode2
0103 CD3201 5      call    cls      ;clear the screen what else
0106 CD6C01 6      call    sputt    ;enter sprite pattern data
0109 CDA001 7      call    setattr   ;set sprite attribute
010C CDBC01 8      call    animate
010F C9      9      ret
0110      10 g2init:
0110 212101 11      ld      hl,regset ;register values
0113 018000 12      ld      bc,0800h  ;b=number of register c=reg no +00h bit 7 set
0116      13 stlp
0116 7E      14      ld      a,(hl)   ;get data value to send
0117 D309    15      out     (9),a
0119 79      16      ld      a,c      ;get register number
011A D309    17      out     (9),a
011C 0C      18      inc     c        ;increment to next register
011D 23      19      inc     hl       ;move hl to next data byte
011E 10F6    20      djnz    stlp      ;do it until all data sent
0120 C9      21      ret
0121      22 regset:
0121 02E10FFF 23      db      2,0e1h,0fh,0ffh,03,7eh,07,0ah
0123      24 addout:
0123      25      ld      a,l        ;we've been here before lsb address
0124 D309    26      out     (9),a      ;send it vdp
0125 7C      27      ld      a,h        ;get msb address
0126 40      28      or      40h       ;make sure vdp knows its a write
012F D309    29      out     (9),a      ;send address to vdp
0131 C9      30      ret
0132      31 cls:
0132 210038 32      ld      hl,3800h    ;start of pattern name table
0135 CD2901 33      call    addout    ;out to vdp
0138 0E03    34      ld      c,3
013A      35 cls1:
013A 06FF    36      ld      b,255      ;loop counter
013C      37 cls2:
013C 78      38      ld      a,b
013D D308    39      out     (8),a      ;out to Vram
013F 07      40      rlca
0140 0F      41      rrca        ;give vdp a rest
0141 10F9    42      djnz    cls2      ;255 times ... remember?
0143 03      43      dec     c        ;*3
0144 20F4    44      jr      nz,cls1
0146 210000 45      ld      hl,0        ;start of pattern table
0149 CD2901 46      call    addout    ;send it to vdp
014C 010018 47      ld      bc,1800h    ;length of pattern table
014F      48 cls3:
014F 0F      49      xor     a        ;reset value
0150 D308    50      out     (8),a      ;send it to vram
0152 07      51      rlca
0153 0F      52      rrca        ;favourite delay
0154 0B      53      dec     bc       ;decrement loop counter
0155 78      54      ld      a,b      ;load it into test flags
0156 01      55      or      c        ;zero
0157 20F6    56      jr      nz,cls3    ;no so carry on
0159 C8EC    57      set     5,h      ;set bit 5 high byte in pattern table

```


THE SOURCE
CHAPTER ELEVEN

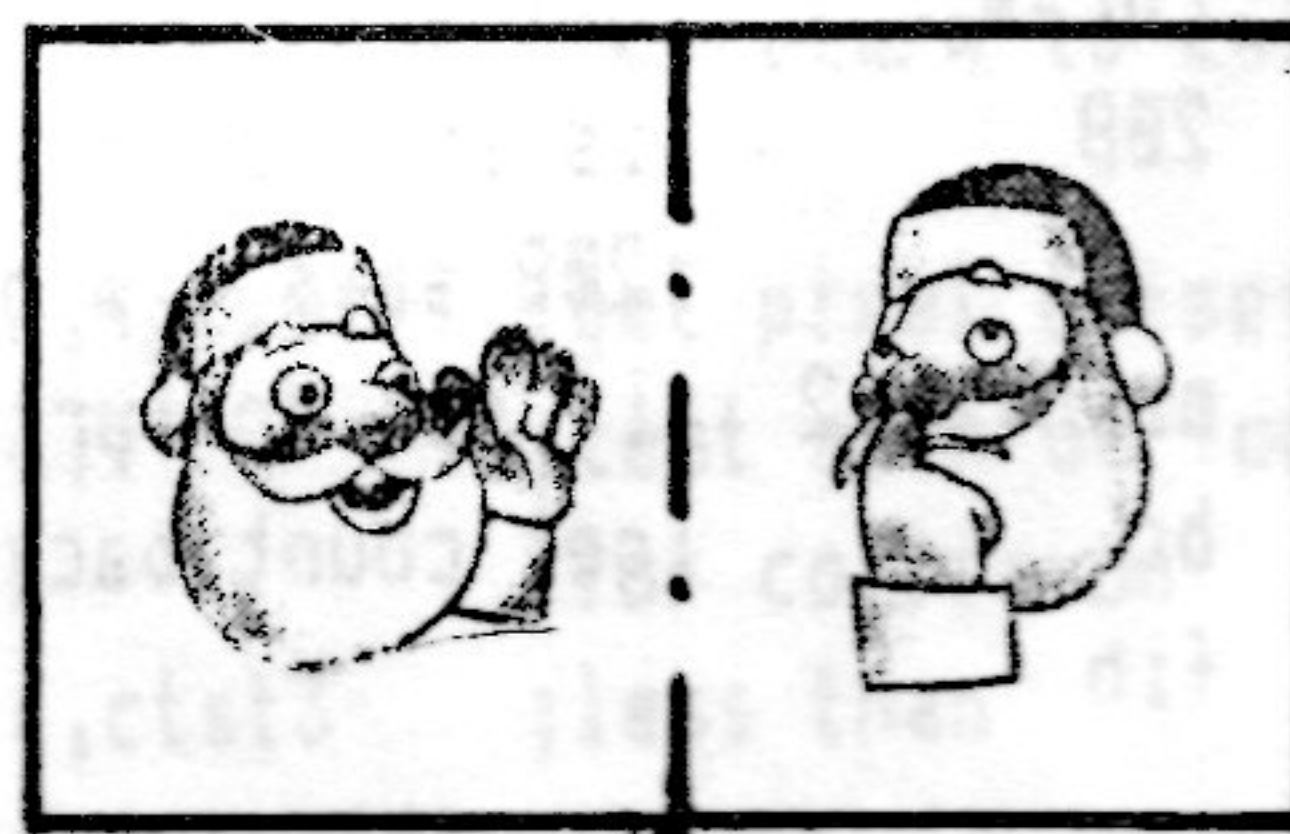
015B CD2901	58	call	addout	;address to give colour table		
015E 010018	59	ld	bc,1800h	;length of colour table		
0161	60	cls4				
0161 3EA1	61	ld	a,0a1h	;yellow foreground/black background		
0163 D308	62	out	(8),a	;and into vram		
0165 0F	63	rrca				
0166 07	64	rlca				
0167 08	65	dec	bc			
0168 78	66	ld	a,b			
0169 B1	67	or	c			
016A 20F5	68	jr	nz,cls4			
016C	69	spput				
016C 210038	70	ld	hl,3800h			
016F CD2901	71	call	addout			
0172 218001	72	ld	hl,spdata			
0175 0620	73	ld	b,20h	;this is the number of bytes		
0177	74	spput1				
0177 7E	75	ld	a,(hl)	;to make up the pattern of each sprite		
0178 23	76	inc	hl	;which is the fed into sprite patt table		
0179 D308	77	out	(8),a			
017B 07	78	rlca				
017C 0F	79	rrca				
017D 10F8	80	djnz	spput1			
017F C9	81	ret				
0180	82	spdata				
0180 00000018	83	db	0,0,0,18h,18h,0,0,0;sprite 0			
0180 00003C3C	84	db	0,0,3ch,3ch,3ch,3ch,0,0;sprite 1			
0190 007E7E7E	85	db	0,7eh,7eh,7eh,7eh,7eh,0;sprite 2			
0198 FFFFFFFF	86	db	0ffh,0ffh,0ffh,0ffh,0ffh,0ffh,0ffh,0ffh;sprite 3			
01A0	87	setattr				
01A0 21003F	88	ld	hl,3f00h	;start of attribute table in vram		
01A3 CD2901	89	call	addout	;send it to vdp		
01A6 3E5E	90	ld	a,94	;y position	117	inc a
01A8 D308	91	out	(8),a	;send it	118	cp 4
01AA 07	92	rlca			119	jr nz,morani
01AB 0F	93	rrca		;wait a bit	120	xor a
01AC 3E7E	94	ld	a,126	;x position	121	morani
01AE D308	95	out	(8),a		122	ld (anipos),a
01B0 07	96	rlca			123	jr animate
01B1 0F	97	rrca			124	anipos
01B2 AF	98	xor	a	;clear a reg	125	db 0
01B3 D308	99	out	(8),a	;pattern number zero to start	126	anitable
01B5 07	100	rlca			127	db 0,1,2,3
01B6 0F	101	rrca			128	delay
01B7 3E0A	102	ld	a,0ah	;colour sprite yellow	129	ld bc,9000h
01B9 D308	103	out	(8),a		130	delay1
01BB C9	104	ret			131	dec bc
01BC	105	animate			132	ld a,b
01BC CDE501	106	call	delay	;if animation too slow change val	133	or c
01BF 21023F	107	ld	hl,3f02h	;pattern no location in vram	134	jr nz,delay1
01C2 CD2901	108	call	addout		135	ret
01C5 3AE001	109	ld	a,(anipos)	;get animation sequence	136	end
01C8 5F	110	ld	e,a			
01C9 1600	111	ld	d,0			
01CB 21E101	112	ld	hl,anitable			
01CE 19	113	add	hl,de			
01CF 7E	114	ld	a,(hl)			
01D0 D308	115	out	(8),a			
01D2 3AE001	116	ld	a,(anipos)			

SPRITE COLLISIONS

Sprite collisions, for the uninitiated, are a 'pain in the butt'. The VDP contains a Sprite Coincidence Flag - bit 6 in the VDP read only register. Whenever two sprites collide this bit is set. Most of you who have tried to utilise this register from Basic will realise that it isn't much good because it doesn't tell you which sprites have collided. And, furthermore, it doesn't necessarily detect the sprite shape because it scans all the pixels with the sprite block - yes, even those that may be transparent. It also detects sprite collisions between sprites that are not actually on the screen if their X and Y co-ordinates coincide.

A sprite impact detection routine should satisfy the following conditions:-

- a) Any sprites having co-ordinates of 0,192 which means that they are not displayed (see previous paragraphs) should be ignored by the impact routine.
- b) The routine should check for coincidence on pixels that are set and not those that are of a transparent nature.
- c) The routine should be flexible enough to cater for 8*8 or 16*16 sprites.



COLLISION DETECTED HERE

Pass 1 errors: 00

```

0100      1 ;Sprite colision detection on specified bounds see text
0100      2 ;Sprite impact test routine to detect collision
0100      3 ;between sprite 0 and any other active sprite.
0100      4 ;Sprite ATTRIBUTE TABLE location held in ATT routine EXITS
0100      5 ;with IMP = 0 --> NO IMPACT with TESTSPR
0100      6 ;
3F00      7 att      equ      3f00h      ;sprite attribute address
0100      8
0100      9      org      100h      ;no need to org include in main code
0100     10
0100     11 ;
0100     12 ;
0100     13
0100     14 dtect      ;Name of subroutine
0100 F5     15      push      af
0101 C5     16      push      bc
0102 D5     17      push      de
0103 E5     18      push      hl      ;preserve all registers
0104 21003F 19      ld      hl,att      ;get attribute address
0107 CDAF01 20      call     rvrarn      ;send address to read routine
010A 5F     21      ld      e,a      ;get first co-ord from attribute table
010B       22      ;and save it in e
010B CDB701 23      call     rdram      ;do another sequential read
010E 57     24      ld      d,a      ;store this in D
010F 7B     25      ld      a,e      ;now we are going to test if
0110 FEC0   26      cp      192      ;if sprite is actually on screen
0112 2004   27      jr      nz,nxtst      ;if it is go on check further
0114 7A     28      ld      a,d      ;if 192 make sure x pos = 0
0115 FE00   29      cp      0      ;so no use testing inactive sprite
0117 C8     30      ret      z      ;so return with imp = 0 no col
0118       31 nxtst
0118 CDB701 32      call     rdram      ;we skip next two bytes by
011B CDB701 33      call     rdram      ;two sequential reads which incs vram to
011E       34      ;next sprite
011E 061F   35      ld      b,31      ;We have sprite 0 so test all others with it
0120       36 nxtst1
0120 C5     37      push      bc      ;save loop counter
0121 CDB701 38      call     rdram      ;get y co-ord
0124 FED0   39      cp      208      ;is it end of active user sprites (Locked off
0126       40      ;208 says I'm not using any more sprites
0126 2004   41      jr      nz,nxtst2      ;if not do further tests
0128 C1     42      pop      bc      ;get count back and go to exit all done
0129 C37501 43      jp      fin
012C       44 nxtst2
012C 6F     45      ld      l,a      ;l = y co-ord
012D CDB701 46      call     rdram      ;now read X co-ord
0130 67     47      ld      h,a      ;store in h
0131 CD7A01 48      call     coltst      ;go check for impact detection
0134 3AB001 49      ld      a,(imp)      ;check if imp = 0 cos if it does there
0137 FE00   50      cp      0      ;has been no collision
0139 200C   51      jr      nz,nxtst3      ;so carry on testing other sprites
013B CDB701 52      call     rdram
013E CDB701 53      call     rdram      ;by skipping to next co-ords
0141 C1     54      pop      bc      ;get loop counter back
0142 10DC   55      djnz     nxtst1      ;if not zero do it all again
0144 C37501 56      jp      fin      ;else go directly to exit routine
0147       57 nxtst3

```



```

0147 C1      58      pop    bc      ;loop counter back
0148 FD218D01 59      ld      iy,sprdat ;iy points to data store
014C FD7203   60      ld      (iy+03),d ;=x co-ord offending sprite
014F FD7304   61      ld      (iy+04),e ;=y co-ord offending sprite
0152 CDB701   62      call   rdram ;now get impact sprite's pattern
0155 FD7705   63      ld      (iy+05),a ;and store it
0158 CDB701   64      call   rdram ;now get colour
015B FD7706   65      ld      (iy+06),a ;and store it
015E 3E20     66      ld      a,32      ;a=number of sprites = 32
0160 90       67      sub     b      ;a-loop counter = impact sprite number
0161 FD7700   68      ld      (iy+00),a ;store it in table
0164         69
0164         70 ;we can now determine where in the attribute table the sprite
0164         71 ;is located
0164         72
0164 CB27      73      sla     a      ;*2
0166 CB27      74      sla     a      ;*4
0168 5F       75      ld      e,a
0169 1600      76      ld      d,0
016B 21003F    77      ld      hl,att ;add it to start of attribute table
016E 19       78      add     hl,de
016F FD7500    79      ld      (iy+00),l ;vram address lsb
0172 FD7401    80      ld      (iy+01),h ;vram address msb
0175         81 fin
0175 E1       82      pop     hl
0176 D1       83      pop     de
0177 C1       84      pop     bc
0178 F1       85      pop     af
0179 C9       86      ret      ;retrieve old register values and ret
017A         87 coltst
017A 3E06      88      ld      a,6      ;set impact detection to 6 pixels in
017C 32BA01    89      ld      (pixoff),a ;store it in pixel offset
017F 3E00      90      ld      a,0      ;now make sure that
0181 32BB01    91      ld      (imp),a ;impct flag = 0= no collision
0184         92 ctst
0184 7D       93      ld      a,l      ;check if this sprite off screen
0185 FEC0      94      cp      192
0187 2004      95      jr      nz,ctst2 ;no its not so do other tests
0189 7C       96      ld      a,h      ;now test x to make sure really off screen
018A FE00      97      cp      0      ;if it is no use going any further
018C C8       98      ret      z      ;so return to caller
018D         99 ctst2
018D 3ABA01   100      ld      a,(pixoff) ;get pixel offset
0190 85       101      add     a,l      ;test each of four sides
0191 BB       102      cp      e      ;for collision
0192 381A      103      jr      c,ctst3 ;less than
0194 3ABA01   104      ld      a,(pixoff)
0197 83       105      add     a,e
0198 BD       106      cp      l
0199 3813      107      jr      c,ctst3
019B 3ABA01   108      ld      a,(pixoff)
019E 84       109      add     a,h
019F BA       110      cp      d
01A0 380C      111      jr      c,ctst3
01A2 3ABA01   112      ld      a,(pixoff)
01A5 82       113      add     a,d
01A6 BC       114      cp      h
01A7 3805      115      jr      c,ctst3
01A9 3E01     116      ld      a,1      ;collision detected

```



```

01AB 320001 117      ld      (imp),a      ;flag it
01AE        118 ctst3
01AE C9     119      ret              ;return calling routine
01AF        120
01AF        121 ;You could actually carry out further tests here before returning. However
01AF        122 ;if you are using interrupts make sure everything can be accomplished in
01AF        123 ;1/50th of a sec. Using ints means you MUST blank off your unused sprites
01AF        124 ;as the time taken to check every sprite would be out of bounds. To check
01AF        125 ;all sprites under interrupt you would have to check half one int etc.
01AF        126
01AF        127 rvrn
01AF 7D     128      ld      a,l
01B0 D309   129      out     (09),a
01B2 7C     130      ld      a,h
01B3 E63F   131      and     3fh      ;make sure vdp knows it has to read
01B5 D309   132      out     (09),a
01B7        133
01B7        134 rdram
01B7 DB08   135      in      a,(8)      ;read from vram
01B9 C9     136      ret
01BA        137 pixoff
01BA 00     138      db      0
01BB        139 imp
01BB 00     140      db      0
01BC        141 tstspr
01BC 00     142      db      0
01BD        143 sprdat
01BD 00000000 144      db      0,0,0,0,0,0,0,0
01C4        145      end

```

The above routine checks for any co-incidence between sprite 0 and any other active sprite. An impact is flagged in IMP.

```

IF    IMP    =    0    then no collision
IF    IMP    =    1    Collision detected

```

The routine is very flexible. By altering PIXOFF you can check for collision on any pixel boundary and can be used with 8*8 or 16*16 sprites. PIXOFF can be dynamically changed from within your program.

If a collision has been detected IMP will contain 1. The offending sprite can then be interrogated or changed by using the table SPRDAT which contains the following information:-

SPRDAT:	SPRITE NUMBER
SPRDAT+1	VRAM ATTRIBUTE TABLE ADDRESS (2bytes)
SPRDAT+3	X CO-ORD
SPRDAT+4	Y CO-ORD
SPRDAT+5	SPRITE PATTERN NUMBER
SPRDAT+6	SPRITE COLOUR

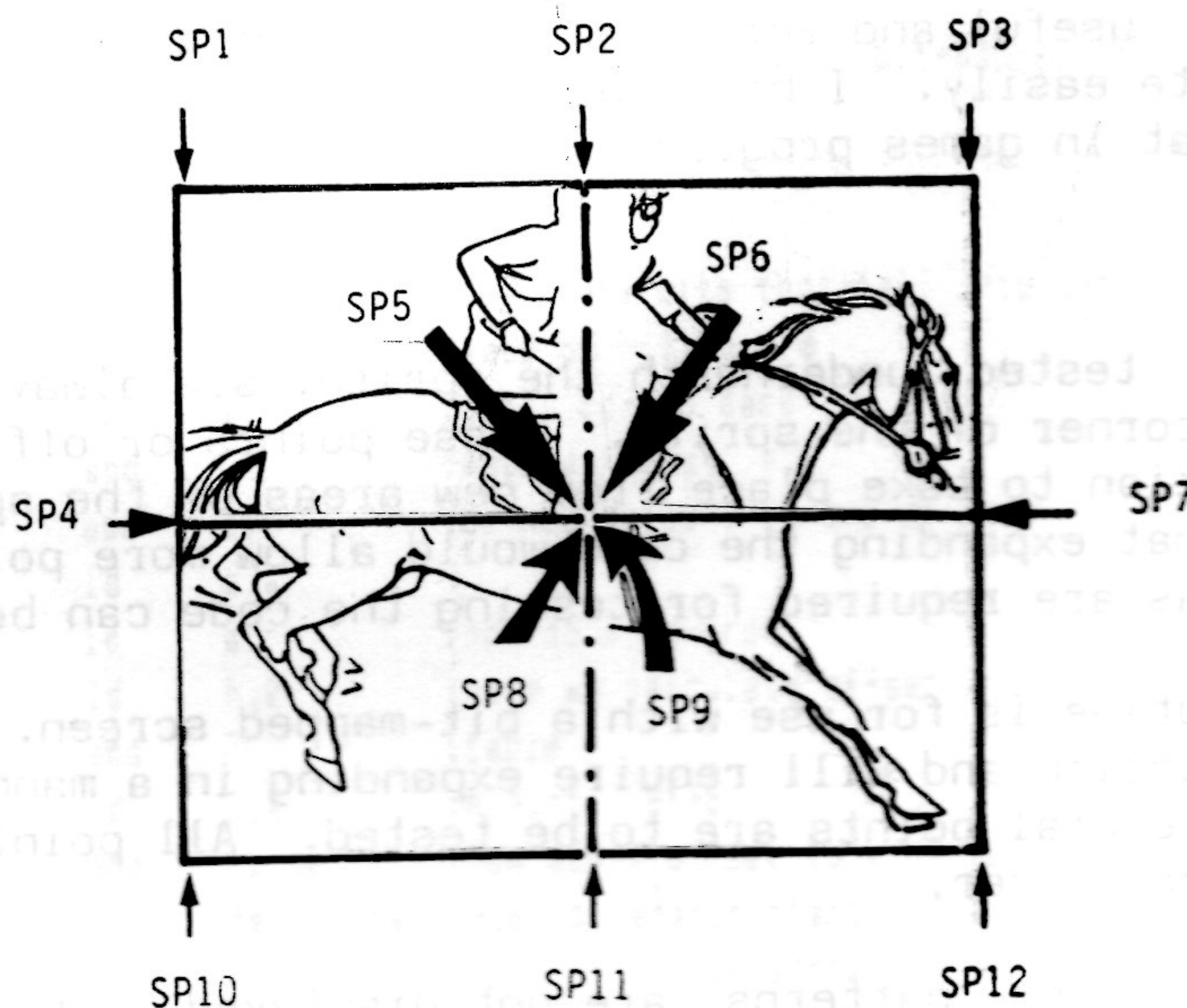
SPRITE COLLISIONS WITH THE PATTERN PLANE

Sometimes it is desirable to be able to detect sprite collisions with the Pattern Plane. For instance: a man-sprite walking through the door of a house. In this situation it is necessary to be able to detect when the sprite is positioned in front of the door - he would look silly walking through the wall!

We have already discussed how Graphic Mode 2 can be setup as a bit-mapped screen or as a character mapped display which means that two separate routines are necessary to cater for each mode.

The character mapped display utilises 8*8 character patterns which are repeated in three equal sections of the Generator Table. The sprite collision routine that uses this mode will use the Generator Table as a look-up table and return a character code at the sprite impact position.

CHARACTER MAPPED SCREEN



HOW THE PROGRAM CHECK CO-INCIDENCE WITH BACKGROUND

The above routine peeks at twelve points within a 16*16 sprite pattern. It then goes on to return the character value from underneath the sprite and also stores the patterns screen address.

TLFT	Returns with screen address of character in SPBUF 5
TRGHT	Returns with screen address of character in SPBUF 6
BLFT	Returns with screen address of character in SPBUF 8
BRGHT	Returns with screen address of character in SPBUF 9

SPBUF 1	Holds character under	SP1
SPBUF 2	Holds character under	SP2
SPBUF 3	Holds character under	SP3
SPBUF 4	Holds character under	SP4
SPBUF 5	Holds character under	SP5
SPBUF 6	Holds character under	SP6
SPBUF 7	Holds character under	SP7
SPBUF 8	Holds character under	SP8
SPBUF 9	Holds character under	SP9
SPBUF 0A	Holds character under	SP10
SPBUF 0B	Holds character under	SP11
SPBUF 0C	Holds character under	SP12

Register DE must point to the sprites attributes (in Vram) on entry to the machine.

This is a useful and very powerful routine that can be adapted for 8*8 sprites quite easily. I have chosen 16*16 sprites because this is the most common format in games programming.

The points tested, underneath the sprite, are always relative to the top left-hand corner of the sprite. These points or offsets can be altered to allow detection to take place from new areas of the sprite. It should also be noted that expanding the code would allow more points to be tested. If smaller areas are required for testing the code can be shortened.

The next routine is for use with a bit-mapped screen. It is presented in a shortened version and will require expanding in a manner similar to the last routine if several points are to be tested. All points are relative to the top left-hand corner.

Because character patterns are not displayed in the normal manner (see chapter on VDP) the routine tests bit-patterns, and instead of peeking the screen it tests coincidence in the Generator Table.

On return from testing impact the A register will be 1 if coincidence has

occurred or Zero if no coincidence. Register DE will hold Vram address of the bit tested.

Again, you can test multiple points by extending the code and further use of SPBUF1-12 may be implemented by setting a bit in the relative buffer if coincidence has been detected.

Study the former listing and have a try at extending this next listing to test for multiple points under the sprite.

All the routines can be interrupt driven. For instance, you could call the coincidence routines on every interrupt and check the buffers, from the main program, when you require knowledge of a collision.

```

0100      1 ;SPRITE 2 ROUTINE to test impact between bit-mapped backdrop and sprite
0100      2 ;Entry conditions: H = Y co-ord & L = X co-ord of sprite
0100      3 ;Exit conditions: DE = bit position in graphic generator & A register =
0100      4 ;if collision or = 0 no collision
0100      5 tsdot ;this is the name of the routine
0100 EE      6      push    hl
0101 70      7      ld      a,h ;get Y co-ord and then calculate relative offset
0102 1600    8      ld      d,0 ;into pattern generator by taking start address
0104 16F0    9      and     240 ;and do the following calcs see vdp chapter
0106 FF     10     ld      e,a
0107 2605    11     ld      b,5 ;loop counter to do it five times when - - - spec
0109         12
0109         13 ;DE = int(y/8)*256+iv- int y/8)*8 see vdp chapter bit-mapped section
0109         14
0109         15 tsdot
0109 CB23    16     sla     e ;*2
010B CB12    17     rl      c ;catch any bits that fall into carry flag
010D 10FA    18     djnz    tsdot ;keep on till b = 0
010F 71      19     ld      a,h ;get Y co-ord back
0110 E607    20     and     7 ;AND mask must = 7
0112 81      21     add     a,e ;de now holds Y co-ord offset
0113 FF      22     ld      e,a
0114 71      23     ld      a,h ;now do X co-ord
0115 2600    24     ld      h,0 ;here we calculate offset into generator
0117 E607    25     and     7 ;table
0119 FF      26     ld      h,a ;hl=int(y/8)*8
011A 19      27     add     hl,de ;now add Y offset to X offset
011B 815B4D01 28     ld      de,gentab ;get generator start address
011F 79      29     add     hl,de ;hl = true Vram address of dot
0120 EE      30     ex      de,hl ;now swap registers to save result
0121 E1      31     pop     hl ;get co-ords back
0122 15      32     push    de ;save vram address
0123 71      33     ld      a,h ;this is where we do BIT TEST
0124 E6FB    34     and     240
0126 FF      35     ld      l,a
0127 2600    36     ld      h,0 ;by finding offset in bit-table
0129 79      37     add     hl,de

```



```

012A      38 ;If you want to know why these values are in bit-table see section on
012A      39 ;calculating addresses in Vram
012A 7E    40      ld      a,(hl)
012B 4F    41      ld      c,a      ;c now =bit pattern from table
012C D1    42      pop     de      ;address of vram back
012D EB    43      ex      de,hl    ;make sure its in HL before going to read vram
012E CD3901 44      call   rvram
0131 A1    45      and     c      ;and with bit pattern
0132 FE00  46      cp      0      ;if zero no collision so exit
0134 CB    47      ret
0135 3E01  48      ld      a,1      ;set collision flag
0137 EB    49      ex      de,hl    ;make sure de = vram address
0138 C9    50      ret          ;return to caller
0139      51 rvram
0139 7D    52      ld      a,1
013A D309  53      out     (9),a
013C 7C    54      ld      a,h
013D E63F  55      and     3fh
013F D309  56      out     (9),a
0141      57 rdram
0141 DE08  58      in      a,8
0143 C9    59      ret
0144      60 bitable
0144 80402010 61      db      128,64,32,16,8,4,2,1,0
014D      62 gentab
014D 0000  63      dw      0000h
014F      64      end
Pass 2 errors: 00

```

Symbol table used: 00K out of 16K.

CHAPTER TWELVE

INTERRUPTS AND THE ZILOG CTC

The Zilog Counter Timer Circuit handles all interrupts on the Einstein range of computers. Unfortunately, on Albert, it is not as useful as it could have been because the manufacturers saw fit not to "hard wire" it to the Vdp interrupt - the Vdp sends out an interrupt flag every time it reaches the bottom of the screen, and if the Ctc had been wired directly to this interrupt pin, on the Vdp, any interrupt would have been synchronised to the vertical blanking period which would have made life a lot easier. However, all is not lost, but synchronisation does suffer.

The features of the CTC are as follows :-

4 independently programmable counter/timer circuits

Standard Z80 daisy-chain interrupt structure provides full vectored, prioritised interrupts.

The CTC can generate **MODE 2** interrupts from any of its four independently programmable channels. It can act as a **timer** or **counter** working with the Z80 clock or an external trigger.

CTC operations are controlled by addressing four Einstein ports - one for each channel.

THE SOURCE CHAPTER TWELVE

PORT	CHANNEL	VECTOR ADDRESS
&28	0	&FB00
&29	1	&FB02
&30	2	&FB04
&31	3	&FB06

The vector addresses are those used by Mos, you can, of course, place these vectors anywhere as long as the fall on eight byte boundaries. It is advisable to place your routine addresses in the original vector addresses, but not necessary.

TIME CONSTANT

When the counter/timer channel is programmed, the **time constant register** receives and stores the value - which can be in the range 1 - 256 [0 = 256]. The constant is then loaded into the down-counter when the counter channel is initialised, and subsequently whenever the count reaches zero.

PRESCALER

The prescaler, is only ever used in the timer mode. It divides the system clock frequency by a factor of 16 or 256.

DOWN COUNTER

Before each count cycle the down-counter is loaded with the time constant register contents, and is then decremented. The Z80 can read the remaining count at any time by reading the CTC port address. When the counter reaches zero, if interrupts are enabled, an interrupt request will be triggered.

PROGRAMMING THE CTC

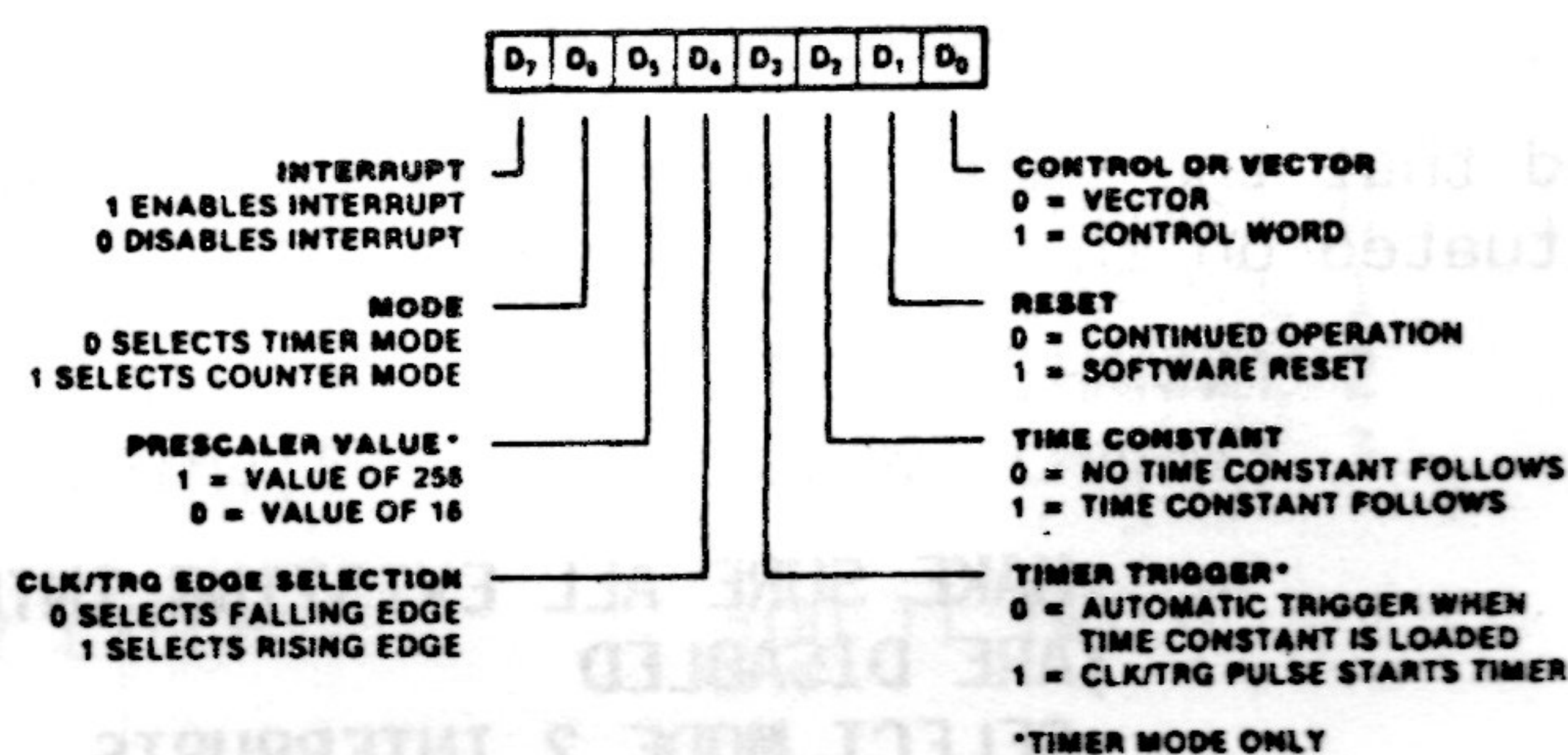
The first step in programming the CTC is to send to the channel, being programmed, a **channel control word**. This word is made up as follows :-

- Bit 0 1 = Control Word
 0 = Vector
- Bit 1 1 = Software reset
 0 = Continued operation

- Bit 2 1 = No time constant to follow
 0 = Time constant to follow
- Bit 3 1 = Clock/Trigger puls starts timer
 0 = Start on receiving time constant
- Bit 4 1 = Trigger on raising edge
 0 = Trigger on falling edge
- Bit 5 1 = Prescaler of 256
 0 = Prescaler of 16
- Bit 6 1 = Counter mode
 0 = Timer mode
- Bit 7 1 = Interrupt enabled
 0 = Disabled interrupt

If bit 2 is reset the CTC will consider the next byte it receives to be a time constant. Setting bit 1 causes the CTC to stop what it is doing, and accept the next set of parameters -a new time constant will also be required.

The interrupt vector is identified by the CTC when bit 0 is reset. The interrupt vector is supplied in the five upper bits. Bit 0 must always be zero to allow the CTC to distinguish between a vector and a control word. Bits 1 & 2 are not used and are set or reset by the CTC to identify the channel issuing the interrupt.



The Vector Table must lie on an eight byte boundary. This table contains the JUMP addresses to routines to be carried out during an interrupt.

The CTC can handle daisy-chaining and will deal with interrupts on a strict priority rotation with channel 0 having the highest priority.

THE SOURCE CHAPTER TWELVE

Later, we shall write a routine to set up the CTC and do something useful under an interrupt.

Before we can initialise the CTC we have to shut down any existing interrupts which will not be required. This can be easily achieved by using the routine given below which switches off the CTC interrupts on all channels.

```
CTCOFF:      LD    B,2          ;LOOP COUNTER
             LD    A,3          ;3 = RESET CTC CHANNEL

CTCLP:      OUT   (28H),A       ;RESET CHANNEL 0
             OUT   (29H),A       ;RESET CHANNEL 1
             OUT   (30H),A       ;RESET CHANNEL 2
             OUT   (31H),A       ;RESET CHANNEL 3

             DJNZ  CTCLP        ;DO IT TWICE
```

The reason we post a 3 to each channel twice is because the CTC, under certain circumstances, be expecting the next byte to be a time constant. By looping through the procedure twice we are ensured of eliminating this problem.

Once we have switched off the CTC the next task, in writing an interrupt routine, is to select interrupt mode 2 and set up the vector table by loading the high byte of the vector address into the Z80 I register, and the low byte into of the CTC chip. Finally, the start address of the interrupt servicing routine must be loaded into the vector table bytes 0 and 1.

It must be reiterated that the vector table supplied in the I register and channel 0 must be situated on an 8 byte boundary.

```
SETCTC:      DI              ;MAKE SURE ALL EXISTING INTERRUPTS
             ;ARE DISABLED
             IM    2          ;SELECT MODE 2 INTERRUPTS
             LD    A,FBH      ;HIGH BYTE OF VECTOR TABLE
             LD    I,A        ;NOW PUT IT INTO PAGE SELECT REGISTER
             LD    A,00H      ;LOW BYTE OF VECTOR ADDRESS
             OUT   (28H),A     ;SEND IT TO CHANNEL 0
             LD    HL,ROUT     ;PUT IT IN THE VECTOR TABLE
             LD    (FBOOH),HL
```

Finally, we must re-enable the CTC interrupts on channel 0 and then clear

the VDP interrupt flag (if we are using the ints for screen access).
Reading the VDP resets the interrupt line of the chip.

SETINT:

```
LD      A,C5H      ;SEND CONTROL BYTES ***
OUT     (28H),A    ;CTC
LD      A,0        ;INTERRUPT EVERY TIME
OUT     (28H),A    ;
IN      A,(9)      ;READ VDP REGISTER
EI      ;ENABLE INTERRUPTS
RETI     ;RETURN FROM INTERRUPT SET UP
```

```
*** C5H   = 1100 0101
           = Bit 7 --> interrupt enabled
             bit 6 --> counter mode
             bit 2 --> no time constant
             bit 1 --> continued operation
             bit 0 --> control word
```

The actual matrix for an interrupt routine should be as follows :-

VECTAB:

```
DB      0,0,0,0,0,0,0,0      ;Must be located on 8 byte boundary
```

CTCOFF:

```
LD      B,2        ;LOOP COUNTER
LD      A,3        ;3 = RESET CTC CHANNEL
```

CTCLP:

```
OUT     (28H),A    ;RESET CHANNEL 0
OUT     (29H),A    ;RESET CHANNEL 1
OUT     (30H),A    ;RESET CHANNEL 2
OUT     (31H),A    ;RESET CHANNEL 3
```

```
DJNZ CTCLP        ;DO IT TWICE
```

SETCTC:

```
DI      ;MAKE SURE ALL EXISTING INTERRUPTS
        ;ARE DISABLED
IM      2        ;SELECT MODE 2 INTERRUPTS
LD      A,FBH    ;HIGH BYTE OF VECTOR TABLE
LD      I,A      ;NOW PUT IT INTO PAGE SELECT REGISTER
LD      A,00H    ;LOW BYTE OF VECTOR ADDRESS
OUT     (28H),A  ;SEND IT TO CHANNEL 0
LD      HL,ROUT  ;PUT IT IN THE VECTOR TABLE
LD      (FBOOH),HL
```


THE SOURCE

CHAPTER TWELVE

SETINT:

```
LD    A,C5H           ;SEND CONTROL BYTES ***
OUT   (28H),A         ;CTC
LD    A,0             ;INTERRUPT EVERY TIME
OUT   (28H),A         ;
IN    A,(9)           ;READ VDP REGISTER
EI                     ;ENABLE INTERRUPTS
RETI                  ;RETURN FROM INTERRUPT SET UP
```

INTROUT:

;Your interrupt routine called at every interrupt

```
DI                     ;DISABLE FURTHER INTERRUPTS
PUSH  AF              ;SAVE ANY REGISTER IN USE BY
PUSH  BC              ;MAIN PROGRAM AND USED
PUSH  HL              ;BY INETRRUPT ROUTINE
IN    A,(9)           ;READ VDP REGISTER
BIT   7,A             ;HAS VDP FINISHED RASTER SCAN
JR    NZ,INT1         ;YES 1 = FINISHED SO DO INT ROUTINE
POP   HL              ;OTHERWISE DO NOTHING AND ....
POP   BC
POP   AF
EI                     ;ALLOW FURTHER INTS
RETI                  ;AND RETURN FROM INTERRUPT
```

INT1:

;Your main interrupt routine goes here then terminate with

```
IN    A,(9)           ;CLEAR ANY VDP PENDING FLAGS
POP   HL
POP   BC
POP   AF
EI                     ;ENABLE INTS
RETI                  ;AND RETURN FROM INTERRUPT
```


Pass 1 errors: 00

```

0100      1 ;ROUTINE TO DEMONSTRATE INTERRUPTS ON EINSTEIN
0100      2 ;Try to figure out how this routine works. It will continuously
0100      3 ;print from 0 to 9 in the left hand corner and first of all
0100      4 ;fill the screen with .... then the screen will be filled
0100      5 ;with *. The interrupt routine adds 1 to TEST when TEST = 10
0100      6 ;a '.' or '*' is printed 30h is added to TEST to give a
0100      7 ;printable Ascii character '1 through 9' this is so fast
0100      8 ;that you can't actually read the numbers but it is obvious
0100      9 ;they are changing. Similiar routines can be used to
0100     10 ;test sprite collisions etc. Good luck!
0100     11 ;
0100     12      org      100h      ;run as a cpm transient
0100     13 ;
0028     14 ctc      equ      28h
0100 C31001 15      jp      start
0103     16      ds      05      ;padding to put table on 8 byte boundary
0108     17 intsrvc
0108 B301B301 18      dw      again1,again1,again1,again1
0110     19 start
0110 CD7301 20      call   mode
0113 CDCA01 21      call   cls
0116 CD8C01 22      call   setint
0119     23 again
0119 F3     24      di          ;disable interrupts
011A     25
011A 21001C 26      ld      hl,1c00h ;
011D 7D     27      ld      a,l
011E D309   28      out     (9),a
0120 7C     29      ld      a,h
0121 F640   30      or      40h
0123 D309   31      out     (9),a
0125 21C601 32      ld      hl,test
0128 7E     33      ld      a,(hl)
0129 FE0A   34      cp      10
012B 2807   35      jr      z,reset
012D C630   36      add     a,30h
012F CDC701 37      call   send
0132 183B   38      jr      again2
0134     39 reset
0134 3E00   40      ld      a,0
0136 77     41      ld      (hl),a
0137 2AE501 42      ld      hl,(vramadd)
013A 23     43      inc     hl
013B 11C01F 44      ld      de,1fc0h
013E E5     45      push    hl
013F ED52   46      sbc     hl,de
0141 E1     47      pop     hl
0142 2011   48      jr      nz,reset2
0144 3AE701 49      ld      a,(toggle)
0147 B7     50      or      a
0148 2803   51      jr      z,nowone
014A AF     52      xor     a
014B 1802   53      jr      reset3
014D     54 nowone
014D 3E01   55      ld      a,l
014F     56 reset3
014F 32E701 57      ld      (toggle),a

```


THE SOURCE
CHAPTER TWELVE

0152 21011C	58	ld	hl,1c01h	
0155	59	reset2		
0155 22E501	60	ld	(vramadd),hl	
0158 7D	61	ld	a,l	
0159 D309	62	out	(9),a	
015B 7C	63	ld	a,h	
015C F640	64	or	40h	
015E D309	65	out	(9),a	
0160 3AE701	66	ld	a,(toggle)	
0163 B7	67	or	a	
0164 2804	68	jr	z,dodot	
0166 3E2A	69	ld	a,"*" ;do a star	
0168 1802	70	jr	sendit	
016A	71	dodot		
016A 3E2E	72	ld	a,"."	
016C	73	sendit		
016C CDC701	74	call	send	
016F	75	again2		
016F FB	76	ei		
0170 C31901	77	jp	again	
0173	78	mode		
0173 018008	79	ld	bc,0808h	
0176 218401	80	ld	hl,regset	
0179	81	nod1		
0179 7E	82	ld	a,(hl)	
017A D309	83	out	(9),a	
017C 79	84	ld	a,c	
017D D309	85	out	(9),a	
017F 0C	86	inc	c	
0180 23	87	inc	hl	
0181 10F6	88	djnz	nod1	
0183 C9	89	ret		
0184	90	regset		
0184 00F00700	91	db	0,0F0h,7,0,3,126,7,4ah	
018C	92	setint		
018C F3	93	di	;stop any interrupts for now	
018D 0602	94	ld	b,2 ;do this loop twice so vdp knows its a setup	
018F 3E03	95	ld	a,3 ;clear all channels of ctc	
0191	96	kill		
0191 D328	97	out	(ctc),a	
0193 D329	98	out	(ctc+1),a	
0195 D32A	99	out	(ctc+2),a	
0197 D32B	100	out	(ctc+3),a ;all channels now done	
0199 10F6	101	djnz	kill	
019B ED5E	102	im	2 ;we want interrupt mode 2	
019D 210801	103	ld	hl,intsrv ;our interrupt routine must be on	
01A0	104		;an eight byte boundary	


```

01A0 7C      105      ld      a,h
01A1 ED4     106      ld      i,a      ;put asb in int vector
01A3 7D      107      ld      a,l
01A4 D328    108      out     (ctc),a      ;lsb into ctc
01A6 3EC5    109      ld      a,0c5h      ;command word see text
01A8 D328    110      out     (ctc),a
01AA 3E00    111      ld      a,0      ;every line
01AC D328    112      out     (ctc),a
01AE DB09    113      in      a,(9)      ;clear vdp int flag
01B0 FB      114      ei      ;enable ints
01B1 ED4D    115      reti     ;return from interrupts
01B3         116 again1

01B3 F3      117      di      ;no interrupts please
01B4 DB09    118      in      a,(9)      ;has flag been set
01B6 CB7F    119      bit     7,a      ;if yes bit 7 will be set
01B8 2003    120      jr      nz,againa ;n
01BA FB      121      ei
01BB ED4D    122      reti
01BD         123
01BD         124 againa
01BD 21C601  125      ld      hl,test      ;THIS IS INT ROUTINE
01C0 34      126      inc     (hl)
01C1 DB09    127      in      a,(9)      ;clear any pending vdp ints
01C3 FB      128      ei
01C4 ED4D    129      reti
01C6         130 test
01C6 00      131      db      00
01C7         132
01C7         133 send
01C7 D308    134      out     (08),a
01C9 C9      135      ret
01CA         136 cls
01CA 21001C  137      ld      hl,1c00h
01CD 7D      138      ld      a,l
01CE D309    139      out     (9),a
01D0 7C      140      ld      a,h
01D1 F640    141      or      40h
01D3 D309    142      out     (9),a
01D5 0E18    143      ld      c,24
01D7         144 cls2
01D7 0628    145      ld      b,40
01D9         146 cls1
01D9 3E20    147      ld      a,20h
01DB D308    148      out     (8),a
01DD 07      149      rlca
01DE 0F      150      rrca
01DF 10F8    151      djnz   cls1
01E1 0D      152      dec     c
01E2 20F3    153      jr      nz,cls2
01E4 C9      154      ret
01E5         155 vramadd
01E5 001C    156      dw      1c00h
01E7         157 toggle
01E7 00      158      db      0
01E8         159      end
Pass 2 errors: 00

```


CHAPTER THIRTEEN

SOUND

All sound on the Einstein is handled by the AY8912 Programmable Sound Generator [PSG]. The PSG is a Large Scale Integrated Circuit [LSI] which can produce a wide variety of sounds under software control. Once a sound has been sent to the chip, the computer is free to carry out other tasks until such time as the sounds, or registers need updating.

The chip uses three, independently controllable, sound channels: A, B, C which can produce pure tone signals or white noise. The frequency response of the PSG ranges from the sub-audible at the lowest frequency to post-audible at the highest frequency. Some sounds may be beyond reproduction on the inbuilt sound amplifier of the Einstein, and may only become apparent when the system is connected to an external hi-fi system.

The basic blocks, within the PSG, are as follows :-

- Tone Generator:** Produces the tone frequencies for each channel.
- Noise Generator:** Produces random frequency modulated noise.
- Mixers:** These combine the outputs from the Tone Generators with that of the noise generator, and there is one per channel.
- Amplitude:** Provides either fixed level sound or a variable volume (amplitude) sound. The variable sound is controlled by the Envelope Generator

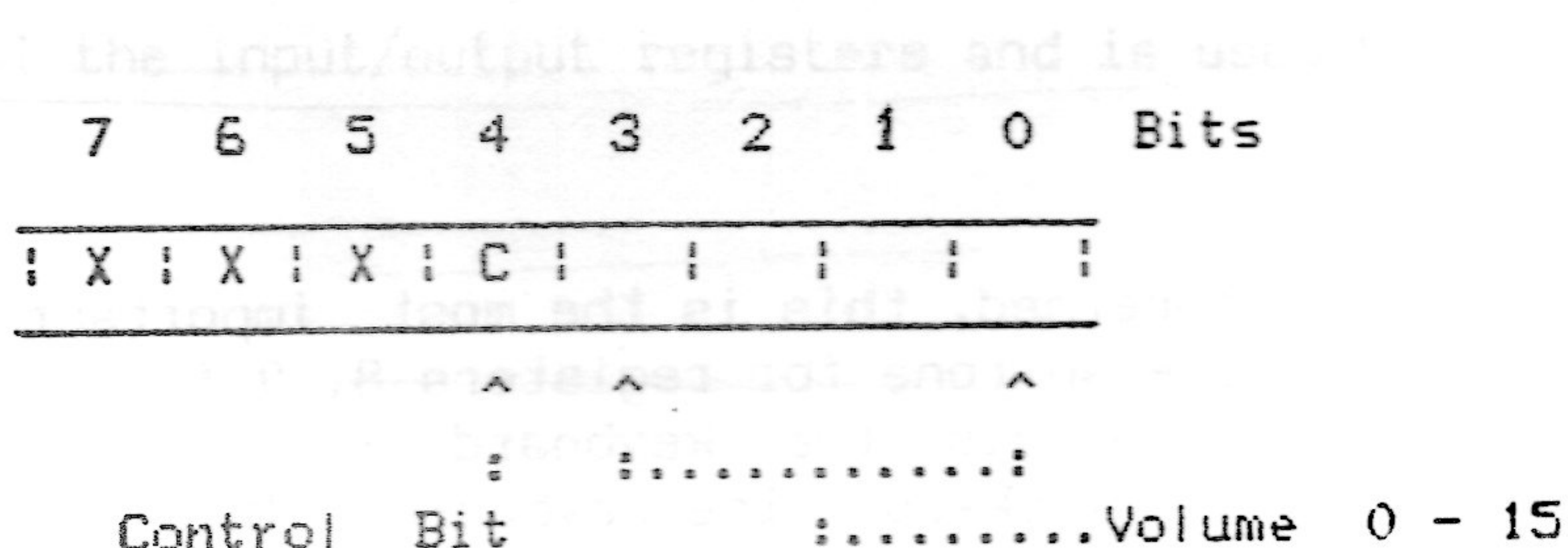
Envelope Generator: Produces an envelope pattern which can be used to amplitude-modulate the output from each of the mixers.

The various operations of the PSG are controlled via 16 Registers R_0 through R_{13} , and their functions are listed below.

R_0	High notes value for channel A
R_1	Low notes value for channel A
R_2	High notes value for channel B
R_3	Low notes value for channel B
R_4	High notes value for channel C
R_5	Low notes value for Channel C
R_6	Noise value
R_7	Mixer channel - Tone ON/OFF Noise ON/OFF I/O Enable
R_8	Volume for channel A
R_9	Volume for channel B
R_{10}	Volume for channel C
R_{11}	High tone envelope period
R_{12}	Low tone envelope period
R_{13}	Envelope shape.

REGISTERS 8-9-10

These registers control the volume of the sound depending on which bits are set. Control of the channels is by 5 **bits only** (0-4) and the remaining three bits are not considered by the PSG.



If bit 4 is set then control is passed to the envelope generator which gives a variable level of sound combined with the differing waveforms.

When bit 4 is 0 control is by the value passed in registers 8, 9, and 10. The explanation is quite simple: if you pass a value between 0 - 15 then the volume is that value with 0 the lowest volume and 15 the highest volume. When the value, in these registers, is 16 the volume varies under control of register 13 - a value of 16 sets bit 4.

REGISTERS 0, 1, 2, 3, 4, 5

These registers control the pitch of the note produced. The lowest channel, of each pair, uses bits 0 - 7 and governs the **Fine Tune** (using just the lower channel produces high pitched notes). The higher channel, of each pair uses bits 0 - 3, and controls the Coarse Tune (low pitched sounds). When the values of the two registers are combined, the result is a 12-bit value, and the note produced is as listed in your Basic manual for that value.

REGISTER 6

Register 6 works in a similar manner to the above registers, but this particular register controls the noise generator. The register only uses 5 bits (0-4) and the higher the value, the lower the resulting frequency of white noise. The range of values that can be input to this register is 0 - 31.

7	6	5	4	3	2	1	0	Bits
x	x	x	0	1	1	1	0	
^			^					
< Not Used >:			:					
			:< 5-Bit Value >:					

REGISTER 7

As far as we are concerned, this is the most important register as this register mixes noise and tone for registers 8, 9 & 10. It also has another **important** role, it controls the keyboard i/o and should any bits be maladjusted it will 'lock-out' the keyboard. This is easy to overcome and is explained later in this chapter.

The register is 'bit specific', and if you study Table 13.1 you should easily understand how different values affect the final output of the three

THE SOURCE CHAPTER THIRTEEN

channels.

7	6	5	4	3	2	1	0	Bit
X	X	1	0	1	1	1	0	
Not : Noise				: Tone				:
used : Enable				: Enable				:
X	X	C	B	A	C	B	A	
< REGISTER 7 BIT ACTIONS >								

BIT SET [1] = OFF BIT RESET [0] = ON

TABLE 13.1

Result	: Channels	: Bits	5	4	3	2	1	0	: Result	: Channels
Noise on	A B C		0	0	0	0	0	0	Tone on	A B C
Noise on	- B C		0	0	1	0	0	1	Tone on	- B C
Noise on	A - C		0	1	0	0	1	0	Tone on	A - C
Noise on	- - C		0	1	1	0	1	1	Tone on	- - C
Noise on	A B -		1	0	0	1	0	0	Tone on	A B -
Noise on	- B -		1	0	1	1	0	1	Tone on	- B -
Noise on	A - -		1	1	0	1	1	0	Tone on	A - -
Noise off	A B C		1	1	1	1	1	1	Tone off	A B C

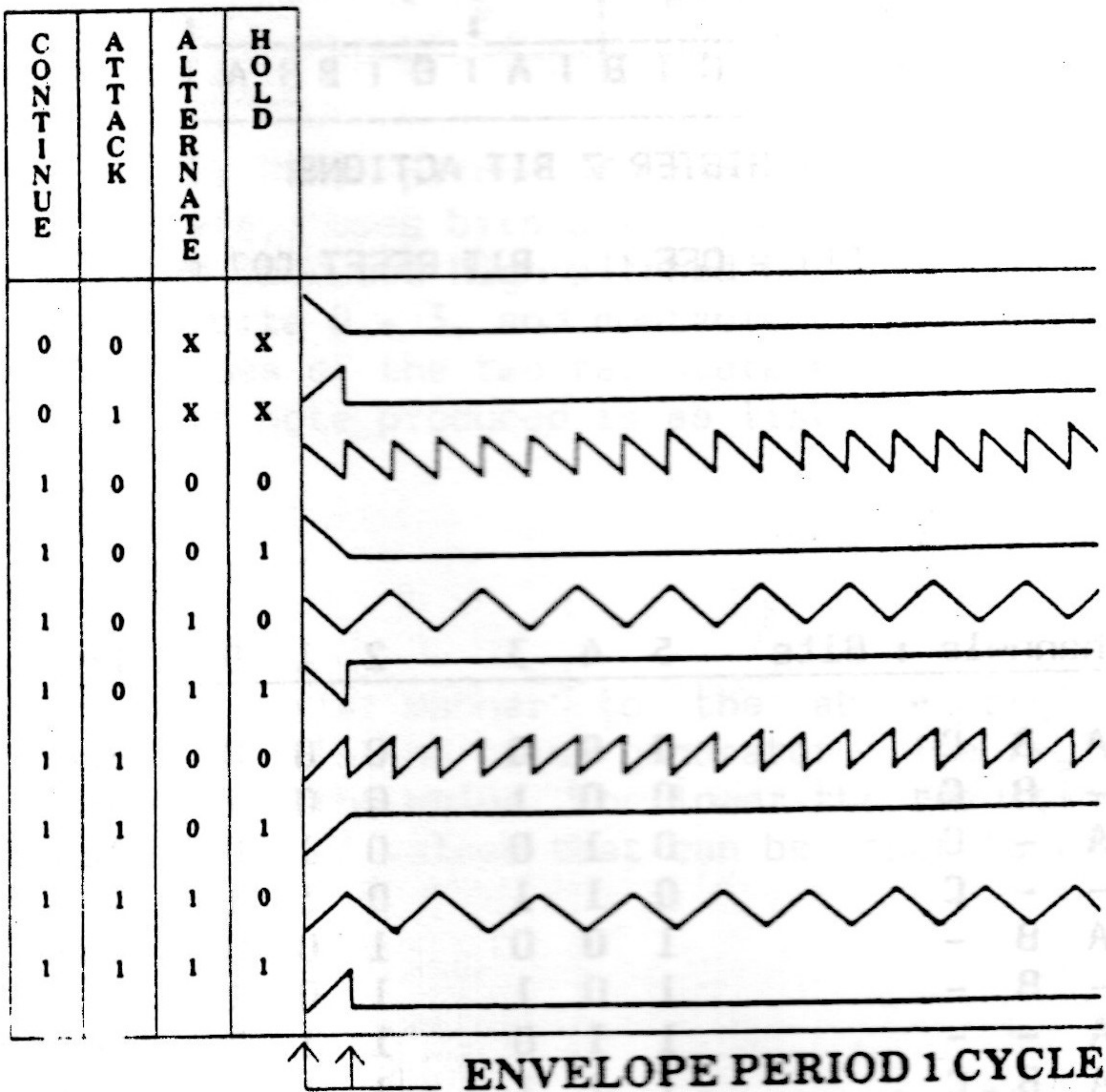
To enable the noise channel B, and the sound on channels A & C, you would need to look in the table and combine the necessary values. E.g. 101010 = 42.

Bits 6 & 7 control the input/output registers and is used by the keyboard.

REGISTER 13

This register uses 4 bits (0-3). The register only affects the overall sound when a value of 16 is placed into registers 8, 9, or 10. Any other value less than 16 will not allow the amplitude to be placed under register 13's control.

ENVELOPE GENERATOR



REGISTERS 11 and 12

Control of the envelope period is by way of these two registers. Combined, the two channels use 16 bits and can have a value in the range 0 - 65535. Register 12 controls the fine tune registers and register 11 the coarse tune registers.

THE SOURCE

CHAPTER THIRTEEN

In a nutshell, these two registers control the time the note is sustained or decayed when under control of the Envelope Generator. Skillful use of these two registers can produce some interesting sound effects ... even speech!

```

0100      1 ;*****
0100      2 ;
0100      3 ;PROGRAM TO DEMONSTRATE SOUND ON THE EINSTEIN USING M/C
0100      4 ;
0100      5 ;*****
0100      6 ;
0100      7 ;
0100      8      ORG      100H
0100      9 ;EQUATES START HERE
0002     10 REGDAT      EQU      2      ;PORTS FOR SENDING
0003     11 DAT         EQU      3      ;RELEVANT DATA
0100     12 ;*****
0100     13 ;SET UP CHANNELS TO AMPLITUDE ONLY BY SENDING A VALUE OF 0-15
0100     14 ;0=LOWEST : 15 = LOUDEST SOUND SO LET'S HAVE SOME umph !
0100     15 ;*****
0100     16 ;
0100 010F08     17      LD      BC,080FH      ;CHANA AMP TO 0FH
0103 CD5601     18      CALL    SEND
0106 010F09     19      LD      BC,090FH      ;CHANB AMP TO 0FH
0109 CD5601     20      CALL    SEND
010C 010C0A     21      LD      BC,0A0CH      ;CHANC AMP TO A
010F         22      ;              LITTLE LESS NOISE
010F CD5601     23      CALL    SEND
0112         24 ;
0112         25 ;*****
0112         26 ;CHANNELS NOW SET ...NOW ENABLE THEM BY SENDING
0112         27 ;RELEVANT DATA TO REGISTER 7
0112         28 ;
0112         29 ;00111000 =38H ENABLES ALL CHANNELS. WE MUST NOW OR 40H
0112         30 ;TO MAKE SURE BIT 6 IS SET OR WE COULD DISABLE THE KEYBOARD!
0112         31 ;*****
0112         32 ;
0112 3E38       33      LD      A,38H          ;00111000
0114 F640       34      OR      40H          ;=01111000
0116 4F         35      LD      C,A          ; DATA IN C REG
0117 0607       36      LD      B,07          ; REGISTER NUMBER 7
0119 CD5601     37      CALL    SEND          ; GO TRIGGER IT
011C         38 ;*****
011C         39 ;REGISTERS NOW SET UP MAIN LOOP TO SEND TUNE DATA HL=>TO DATA
011C         40 ;REGISTERS 0-5 USED FOR SENDING TUNE DATA 0-3 FOR TUNE
011C         41 ; 4&5 FOR BASS NOTE
011C         42 ;*****

```



```

011C 43 ;
011C 44 MAIN
011C 215D01 45 LD HL,MUSIC ;POINT HL TO DATA
011F 46 LOOP
011F 4E 47 LD C,(HL) ;NOTE INTO C
0120 79 48 LD A,C ; PUT IN A FOR
0121 FEFF 49 CP 0FFH ;TEST FOR END OF DATA
0123 282F 50 JR Z,FIN ;IF YES GO TO FINISH
0125 0600 51 LD B,0 ; FINE TUNE REG CHANA
0127 CD5601 52 CALL SEND ; SEND REG AND NOTE
012A 0602 53 LD B,02 ;FINE TUNE CHANB
012C CD5601 54 CALL SEND ; SEND SAME NOTE
012F 23 55 INC HL ;POINT TO NEXT DATA
0130 4E 56 LD C,(HL) ;GET NEXT NOTE
0131 2501 57 LD B,01 ;COARSE TUNE REG CHANA

0137 CD5601 58 CALL SEND
013B 0601 59 LD B,03 ;COARSE TUNE REG CHANB
013E CD5601 60 CALL SEND
0140 23 61 INC HL ; NEXT NOTE
0141 4E 62 LD C,(HL) ;NOTE IN C REG
0143 0601 63 LD B,04 ;FINE TUNE CHANC
0145 CD5601 64 CALL SEND
0147 23 65 INC HL ; NEXT NOTE
0148 4E 66 LD C,(HL) ;NEXT NOTE IN C
014A 0605 67 LD B,05 ;COARSE TUNE REG CHANB
014C CD5601 68 CALL SEND ; MAKE A NOISE
014E 23 69 INC HL ; NEXT NOTE FOR MAIN LOOP
014F 70 ;
014A 71 ;*****
014A 72 ;PUT A DELAY TO ALLOW NOTES TO PLAY A SOME KIND OF TEMPO
014A 73 ;*****
014A 74 ;
014A 75 LD BC,7500H ; COUNTDOWN TEMPO 3R
014D 76 DELY
014D 0B 77 DEC BC
014E 79 78 LD A,C
014F B0 79 OR B ;IF ANY COUNT LEFT
0150 20FB 80 JR NZ,DELY ;ZERO FLAG NOT SET
0152 18CB 81 JR LOOP ;ZERO SO DO IT ALL AGAIN
0154 82 ;
0154 83 ;*****
0154 84 ;JUMPS HERE WHEN DATA = 0FFH SO PUT IT IN A LOOP SO IT
0154 85 ;GOES ON AND ON AND ON AND ON AND ON
0154 86 ;*****
0154 87 FIN
0154 18C6 88 JR MAIN

```


THE SOURCE CHAPTER THIRTEEN

```

0156      84 ;
0156      90 ;*****
0156      91 ;SUBROUTINE TO SEND REGISTER AND DATA TO SOUND CHIP. IT ASSUMES
0156      92 ;REGISTER NUMBER IN B & DATA IN C..
0156      93 ;*****
0156      94 ;
0156      95 SEND
0156 78    96      LD      A,B      ; GET REGISTER
0157 D302  97      OUT     (REGDAT),A
0159 79    98      LD      A,C      ;GET DATA
015A D303  99      OUT     (DAT),A   ; SEND IT!
015C C9   100     RET              ; RETURN TO CALLER
015D      101 ;
015D      102 ;*****
015D      103 ;MUSIC DATA GOES HERE IN THE FORMAT DB MUSIC,BASS,MUSIC,ETC
015D      104 ;*****
015D      105 ;
015D      106 MUSIC
015D 0000DE01 107     DW      00,478,00,379,00,319,00,379,00,239,00
0173 3F010000 108     dw      319,00,379,00,319,00,478,00,379,00,319
0189 00007B01 109     dw      00,379,00,239,00,319,00,379,00,319,00
019F DE010000 110     dw      478,00,379,00,319,00,239,00,319,00,379
01B5 00003F01 111     dw      00,319,00,478,00,379,00,319,00,379,00
01CB EF000000 112     dw      239,00,319,00,379,239,319,239,478,319
01DF 7B01EF00 113     dw      379,239,319,190,239,219,379,190,319,159,239
01F5 EF003F01 114     dw      239,319,119,358,127,284,142,239,159,179,159
020B DE010000 115     dw      478,00,379,00,319,00,239,159,638,179,506,123
0223 AA01FD00 116     dw      426,253,319,319,506,253,426,213,319,179,426

0239 BE00DE01 117     dw      190,478,239,379,142,239,159,319,00,379,00,319
0251 0000EF00 118     dw      00,239,239,319,239,478,319,379,239,319,190,239
0269 EF007B01 119     dw      239,379,190,319,159,239,239,319,119,358,127,284
0281 BE00EF00 120     dw      142,239,159,179,00,478,00,379,00,319,00,239,159
029B 7E02B300 121     dw      638,179,506,123,426,253,319,319,506,253,426,213
02B3 3F01B300 122     dw      319,179,426,239,478,319,379,190,319,239,239,00
02CB 7B010000 123     dw      379,00,319,00,239,239,638,239,956,319,638,239
02E3 DE01BE00 124     dw      478,190,478,239,638,190,638,159,478,239,956,119
02FB 00007F00 125     dw      00,127,716,142,568,159,638,00,956,00,638,00,478
0315 00005303 126     dw      00,851,0FFFFH
031B      127 END
Pass 2 errors: 00

```

Symbol Table used: 00K out of 16K.

CHAPTER FOURTEEN

DISCS

You should be aware that the Einstein is, virtually, CP/m (Control Program Microcomputers), and it would be a good idea to obtain a book on the theory of CP/m.

Programs operating under CP/m use static locations to perform specific tasks. The advantage of this is "portability". A program that works on the Einstein will also work on an Amstrad or another CP/m system.

Mos Calls are available to perform the same tasks but it is always a good idea to use the CP/m protocol for all input/output operations, because this saves a lot of programming when converting from one system to another computer, and CP/m calls are just as easy to use. **"Mastering CPM"** by Alan Miller and **"The CPM Handbook"** by Rodney Zaks are two excellent books on the subject of CP/m.

Disc handling falls into the above category. Obviously, different machines will have varied formats but the calling procedures for accessing files, writing, reading, and creating files are just the same.

The following explanation is based on the CP/m structure. We shall define a series of subroutines with specific functions such as, opening a file etc. These can, at a later date, be slotted into your own programs. Tagged to the front of the subroutines is a program that demonstrates how they are used. A **word of warning** ! Whenever you write routines that use disc i/o always use a disc that doesn't contain valuable information. You can see

THE SOURCE CHAPTER FOURTEEN

the logic, if we designed a format routine and made a mistake and tried it out on a disc that contain lots of file.... I'll leave it to your imagination.

FCB

Every file, on a disc, has, at least, one FCB. The File Control Block defines the size of the file and its position on the disc. A FCB is **thirty six** bytes in length, but for the purpose of this explanation we are only interested in the following :-

Byte 0	Drive Number	0 = Default 1 = Drive 0 2 = Drive 1 etc.
--------	--------------	--

Bytes 1 - 8	FILENAME up to eight characters in length. Must be padded with '0' if less than eight characters are used.
-------------	---

Bytes 9 - 11	FILE EXTENSION TYPE : Xbas Com etc. Padded with zeros if less than three bytes used.
--------------	--

The remaining bytes, within the FCB, must, initially, be set to zero.

To access discs, or any other peripheral, we use a **CALL** to location 5 in low memory. This is one of CP/m's **Basic Disc Operating System (BDOS)** calls. For example: to open a file we load DE with the location, in memory, of the FCB, register C is loaded with the value 15 which is the flag for OPEN FILE, and perform a CALL 5. The command will then be carried out by CP/m. On return the A register will contain -1 (255) if the file is not found.

PROTOCOL "OPEN FILE"

```
LD    DE,FCB
LD    C,15
CALL  5
CP    255      ;Has file been found ?
JP    Z,NOTFOUND
```


CLOSING FILES

Obviously, once we have finished with a particular file we cannot discard it in a nonchalant manner. We must CLOSE it following the correct CP/m procedures. Here again, we must use DE to point to the FCB but, this time, we do not have to clear the block - if we now looked at the FCB area we would find that some of the data had be altered by CP/m during are open, write and read operations, and these will be saved to the disc when we CLOSE the file.

PROTOCOL CLOSING FILES

```
LD    DE,FCB
LD    C,16      (CLOSE FLAG)
CALL  5
```

WRITING TO DISC

Data is written in **128 byte sectors** and to allows us to write these 128 byte blocks we must have a **buffer area** to store the data before we write it to disc.

To put the data on the disc we must, repeatedly, fill the buffer with the data, write 128 bytes to disc, fill buffer with data, write it to disc, and so on until we have stored all the information.

With the routines under discussion, data is stored sequentially on the disc in a way that the first 128 bytes occupy data position 0, the second 128 byte dump occupies data position 1 etc. etc. However, in order to store the data we must, first, **CREATE** a file - we cannot simply open an existing file. The routine **Create** does this for us. This routine will return with the **zero flag set** if there isn't enough space in the **directory area**.

The subroutines are all documented, and they should enable you to understand, and utilise them within your own programs. We have only examined **sequential file access**, **random access** is far more versatile but is considerably more complicated and beyond the scope of this book. However, after understanding the following routines, you should have no difficulty in following a good CP/m book on this subject. However, for the brave, I have included a set of routines that will perform random access.

THE SOURCE CHAPTER FOURTEEN

```

0100      1 ;THE FOLLOWING IS AN EXAMPLE OF HOW TO SEQUENTIALLY ACCESS DISC FILES
0100      2 ;THE SUBROUTINES ARE AT THE END OF THE PROGRAM AND CAN BE INSERTED IN
0100      3 ;ANY OF YOUR OWN ROUTINES ... THE FOLLOWING IS JUST AN AID TO UNDERSTANDING
0100      4 ;HOW TO USE THEM
0100      5 ;
0100      6          org      100h
0100      7
005C      8 fcb          equ      5ch          ;use actual area put aside by cpm
0005      9 bdos         equ      5           ;bdos call 5
0080     10 buff         equ      80h          ;128 bytes
0100     11
0100 C36F01 12          jp          start
0103     13 ;buffer area here
0103     14 name
0103 00544553 15          db          0,"TEST    DAT"
010F 00000000 16          db          0,0,0,0,0,0,0,0,0
0118 00000000 17          db          0,0,0,0,0,0,0,0,0
0121 00000000 18          db          0,0,0,0,0,0,0,0,0;enough space for at least 36 bytes
012A     19
012A     20 rec1
012A 54686973 21          db          "This is record number 1"
0141     22 rec2
0141 54686973 23          db          "This is record number 2"
0158     24 rec3
0158 54686973 25          db          "This is record number 3"
016F     26
016F     27
016F     28 start
016F CD7601 29          call start1
0172 CDE401 30          call read_it
0175 C9     31          ret
0176     32 start1
0176     33
0176     34 ;the following creates a file DO NOT USE THES ROUTINES ON A VALUABLE
0176     35 ;DISC USE A DISC YOU DON'T CARE ABOUT TO TEST ... YOU MAY HAVE MADE
0176     36 ;A MISTAKE AND WE ARE TESTING DISC ACCESS ROUTINES
0176     37 ;
0176 210301 38          ld          hl,name
0179 115C00 39          ld          de,fcb
017C 012400 40          ld          bc,36          ;name is padded to 36 characters
017F ED80 41          ldir          ;move from hl into fcb (de)
0181     42
0181 CDA702 43          call      open          ;open a file with name TEST.DAT
0184 2809 44          jr          z,can_creat;ok so carry on
0186 11D701 45          ld          de,mes1
0189 0E09 46          ld          c,9
018B CD0500 47          call      bdos          ;File already exists
018E C9 48          ret
018F     49
018F     50 can_creat
018F CDCF02 51          call      create          ;ok so create file
0192 CDC901 52          call      clear          ;clear out buffer area
0195 212A01 53          ld          hl,rec1
0198 118000 54          ld          de,buff
019B 011700 55          ld          bc,23
019E ED80 56          ldir          ;move data into buffer
01A0     57

```


THE SOURCE CHAPTER FOURTEEN

```

01A0 CDB802 58      call  write      ;so we can write it to disc
01A3 CDC901 59      call  clear      ;now clear buffer again
01A6 214101 60      ld      hl,rec2
01A9 118000 61      ld      de,buffer
01AC 011700 62      ld      bc,23
01AF EDB0   63      ldir
01B1       64
01B1 CDB802 65      call  write
01B4 CDC901 66      call  clear
01B7       67
01B7 215801 68      ld      hl,rec3
01BA 118000 69      ld      de,buffer
01BD 011700 70      ld      bc,23
01C0 EDB0   71      ldir
01C2       72
01C2 CDB802 73      call  write
01C5 CDB202 74      call  close
01C8 C9     75      ret              ;all done so return
01C9       76
01C9       77 clear
01C9       78 ;*****
01C9       79 ;Flush out buffer so it
01C9       80 ;can be used again .
01C9       81 ;*****
01C9       82
01C9 218000 83      ld      hl,buffer
01CC 3620   84      ld      (hl)," "
01CE 118100 85      ld      de,buffer+1
01D1 018000 86      ld      bc,80h
01D4 EDB0   87      ldir              ;fill with nulls
01D6 C9     88      ret
01D7       89
01D7       90 mes1
01D7 46696C65 91      db      "File Exists"$
01E4       92
01E4       93
01E4       94 ;Here we read the file we have just created and written to
01E4       95
01E4       96 read_it
01E4 210301 97      ld      hl,name      ;pointer to file
01E7 115C00 98      ld      de,fcf      ;point de at fcf
01EA 018000 99      ld      bc,80h      ;128 byte count
01ED EDB0   100     ldir              ;move 128 bytes from name into fcf
01EF       101
01EF       102 ;now open the file
01EF       103
01EF CDA702 104     call  open
01F2 2012   105     jr      nz,ok        ;zero flag not set so open ok
01F4 11FD01 106     ld      de,mes2      ;falls through if can't open
01F7 0E09   107     ld      c,9          ;Bdos call print message
01F9 CD0500 108     call  bdos
01FC C9     109     ret              ;program will crash because no calling
01FD       110     ;program but this is how you would use
01FD       111     ;the check in your own programs.
01FD       112 mes2
01FD 4E4F2046 113     db      "NO File!$";Bdos messages must terminate with '$'
0206       114
0206       115 ok
0206 CDC502 116     call  read      ;now read from file

```


THE SOURCE CHAPTER FOURTEEN

```

0209 2010      117      jr      nz,ok2      ;something wrong so return to caller
020B 210000     118      ld      hl,buffer      ;buffer
020E 112602     119      ld      de,pbuff      ;print buffer

0211 018000     120      ld      bc,80h      ;128 bytes
0214 ED80      121      ldir      ;move the bytes!
0216 CD1F02     122      call    print      ;print the read
0219 18EB      123      jr      ok      ;go back for some more
021B          124      ok2
021B CDB202     125      call    close
021E C9        126      ret
021F          127      print
021F 21A502     128      ld      hl,pbuff2
0222 CBFE      129      set     7,(hl)      ;set bit 7 of last char in print buffer
0224 CF        130      rst      8      ;Mos print call
0225          131
0225 CF        132      db      2c7b      ;Mos function number
0226          133      pbuff
0226          134      ds      77h      ;reserve 127 bytes because the last byte
02A5          135      pbuff2
02A5 00        136      db      0      ;we have to set bit 7 so Mos knows end of
02A6          137      ;print.
02A6 C9        138      ret
02A7          139      ;*****
02A7          140      ;Main routines here
02A7          141      ;*****
02A7          142
02A7          143      open
02A7          144      ;*****
02A7          145      ;Open a file sets flag
02A7          146      ;if file not found
02A7          147      ;*****
02A7          148
02A7 115C00     149      ld      de,fcf
02AA 0E0F      150      ld      di,15
02AC CD0500     151      call    bdos
02AF FEFF      152      cp      255      ;if -1 then file not found so
02B1          153      ;zero flag set on return
02B1 C9        154      ret
02B2          155
02B2          156      close
02B2          157      ;*****
02B2          158      ;Close a file
02B2          159      ;*****
02B2          160
02B2 115C00     161      ld      de,fcf
02B5 0E10      162      ld      c,16      ;function number
02B7 CD0500     163      call    bdos
02BA C9        164      ret

```



```

02B8      166 write
02B8      167 ;*****
02B8      168 ;Write to a file returns
02B8      169 ;NZ if write error
02B8      170 ;*****
02B8      171
02B8 115C00 172      ld      de,fcf
02BE 0E15   173      ld      c,21      ;Bdos function number
02C0 CD0500 174      call   bdos
02C3 B7     175      or      a          ;set or reset Z flag for error

02C4 C9     176      ret

02C5      177
02C5      178 read
02C5      179 ;*****
02C5      180 ;Read from a file
02C5      181 ;Returns NZ if read error
02C5      182 ;*****
02C5      183
02C5 115C00 184      ld      de,fcf
02C8 0E14   185      ld      c,20      ;function number
02CA CD0500 186      call   bdos
02CD B7     187      or      a          ;check for error and set flag
02CE C9     188      ret

02CF      189
02CF      190 create
02CF      191 ;*****
02CF      192 ;Create a file. Z flag set if
02CF      193 ;no directory space available
02CF      194 ;*****
02CF      195
02CF 115C00 196      ld      de,fcf
02D2 0E16   197      ld      c,37      ;function number
02D4 CD0500 198      call   bdos
02D7 FEFF   199      cp      255      ;-1 if fault
02D9 C9     200      ret
02DA      201
02DA      202      end

Pass 2 errors: 00

```


THE SOURCE
CHAPTER FOURTEEN

0080		; BUFF EQU 80H
		; WRITE A RECORD.
		; ON ENTRY:-
		; HL IS RECORD NUMBER
		; BUFFER CONTAINS DATA TO SAVE
		; RETURNS
		; NZ IF READ FAILED
		; Z IF READ OK
		; FIRST OF ALL, CALCULATE
		; THE EXTENT AND RECORD NUMBER
0000'		WRITE:
0000'	7D	LD A,L
0001'	E6 7F	AND 7FH
0003'	32 011F'	LD (NR),A
0006'	29	ADD HL,HL
0007'	7C	LD A,H
0008'	32 00EF'	LD (EXTENT),A
		; NOW ATTEMPT TO OPEN FILE
000B'	11 00E3'	LD DE,FCB
000E'	0E 0F	LD C,15
0010'	CD 0005	CALL 5
0013'	3C	INC A
0014'	20 0C	JR NZ,OPENED
		; CREATE A NEW FILE
0016'	11 00E3'	LD DE,FCB
0019'	0E 16	LD C,16H
001B'	CD 0005	CALL 5
001E'	3C	INC A
001F'	CA 0048'	JP Z,FAILED
0022'		OPENED:
		; NOW SELECT RECORD NUMBER
0022'	3A 011F'	LD A,(NR)
0025'	32 0103'	LD (FCB+32),A
		; COPY DATA FROM BUFFER
0028'	21 004D	LD HL,BUFFER
002B'	11 0080	LD DE,BUFF
002E'	01 0080	LD BC,128
0031'	ED B0	LDIR

0033'	11	00E3'
0036'	0E	15
0038'	CD	0005
003B'	B7	
003C'	20	0A
003E'	11	00E3'
0041'	0E	10
0043'	CD	0005
0046'	AF	
0047'	C9	
0048'		
0048'	3E	01
004A'	CB	47
004C'	C9	
004D'	54	48 49 53
0051'	20	49 53 20
0055'	41	20 54 45
0059'	53	54 20 4D
005D'	45	53 53 41
0061'	47	45
0063'		
00E3'	00	44 41 54
00E7'	41	20 20 20
00EB'	20	44 41 54
00EF'	00	00 00 00
00F3'	00	00 00 00
00F7'	00	00 00 00
00FB'	00	00 00 00
00FF'	00	00 00 00
0103'	00	00 00 00
0107'	00	00 00 00
010B'	00	00 00 00
010F'	00	00 00 00
0113'	00	00 00 00
0117'	00	00 00 00
011B'	00	00 00 00
00EF'		
011F'	00	

[illegible]

THE SOURCE
CHAPTER FOURTEEN

0080		;	;
		BUFF EQU 80H	
		;	
		; READ A RECORD.	
		; ON ENTRY:-	
		; HL IS RECORD NUMBER	
		;	
		; RETURNS	
		;	
		; NZ IF READ FAILED	
		; Z IF READ OK	
		;	
		; ACTUAL DATA IN BUFFER	
		;	
		;	
		; FIRST OF ALL, CALCULATE	
		; THE EXTENT AND RECORD NUMBER	
		;	
0000'		READ:	
0000'	7D	LD A, L	
0001'	E6 7F	AND 7FH	
0003'	32 00FD'	LD (NR), A	
0006'	29	ADD HL, HL	
0007'	7C	LD A, H	
0008'	32 00CD'	LD (EXTENT), A	
		;	
		; NOW ATTEMPT TO OPEN FILE	
		;	
000B'	11 00C1'	LD DE, FCB	
000E'	0E 0F	LD C, 15	
0010'	CD 0005	CALL 5	
		;	
0013'	3C	INC A	
0014'	28 26	JR Z, FAILED	
		;	
		; NOW SELECT RECORD NUMBER	
0016'	3A 00FD'	LD A, (NR)	
0019'	32 00E1'	LD (FCB+32), A	
		;	
		; AND READ DATA	
001C'	11 00C1'	LD DE, FCB	
001F'	0F 14	LD C, 14H	
0021'	CD 0005	CALL 5	
0024'	B7	OF A	
0025'	20 15	JR NZ, FAILED	
		;	
		; COPY TO BUFFER	
		;	
0027'	11 0041'	LD DE, BUFFER	
002A'	21 0080	LD HL, BUFF	
002D'	01 0080	LD BC, 128	
0030'	ED 80	LDIR	
		;	
		; AND CLOSE FILE.	
		;	
0032'	11 00C1'	LD DE, FCB	
0035'	0E 10	LD C, 10H	
0037'	CD 0005	CALL 5	
		;	
003A'	AF	XOR A; SET ZERO CONDITION	
003B'	C9	RET ; RETURN	
		;	
003C'		FAILED:	
		; SET NON ZERO CONDITION	
003C'	3E 01	LD A, 1	
003E'	CB 47	BIT 0, A	
0040'	C9	RET	
0041'		BUFFER: DS 128	
00C1'	00 44 41 54	FCB: DB 0, "DATA DAT"	
00C5'	41 20 20 20		
00C9'	20 44 41 54		
00CD'	00 00 00 00		
00D1'	00 00 00 00		
00D5'	00 00 00 00		
		DB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0	

THE SOURCE
CHAPTER FOURTEEN

```
00D9' 00 00 00 00
00DD' 00 00 00 00
00E1' 00 00 00 00
00E5' 00 00 00 00
00E9' 00 00 00 00
00ED' 00 00 00 00
00F1' 00 00 00 00
00F5' 00 00 00 00
00F9' 00 00 00 00
00CD'
00FD' 00

DB 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
DB 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

EXTENT EQU FCB+12
NR:DB 0
END
```


USEFUL CALLS TO BASIC

02A1 CALL 02A2
 DB ')"

Test for character defined in the DB statement. After the call HL points to character after test.

0287

Return byte value in A.

028A

Return an integer value in A B. High byte in A register.

028D

Return value in FPA (Floating Point Accumulator).

0290

Return expression value in FPA can be used for a string.

The above routines should not be called. They should be used as an exit avenue e.g JP 0290. See Basic modification routine in main book area.

0296

Compare HL with DE.

Returns:	HL = DE	zero flag set
	HL > DE	carry flag set

THE SOURCE APPENDIX A

HL <= DE NC

CALL 0296
JR Z,EQUAL
JR NC,HGREATER

02A9

Increments HL until a non-space character is found. Exits with HL pointing to character. Zero flag set if character = : Carry flag set if 0 - 9 found.

02B8

Print a message routine. High byte of last character must have high-bit set.

CALL 02B8
DB "HELLO ITS WORKIN","G"+80H
RETURNS HERE

033E

Searches for Line Number given in DE. Starts from top of text. Carry flag and zero flag set if line found. BC points to beginning of line, HL points to beginning of NEXT line.

0341

As above but searches from current line number.

0569

Converts a 2-byte integer into a floating point number. Must be called with MSB of integer in A and LSB in B.

2108

Converts a number stored in the FPA into Ascii starting at the position pointed to by HL. Returns with HL pointing to start of Ascii string.

2BF8

Reset vdu - psg - fdc and Basic.

2C61

CLS

2C83

CLS40

2C8B

CLS32

2CA1

Set backdrop colour ... LD A,colour
OR OFOH
CALL 2CA1

2EAF

Set sprite magnification. LD A,param

0 = normal
1 = normal magnified
2 = size 1 sprites
3 = size 1 magnified

35BD

Keyboard scan.

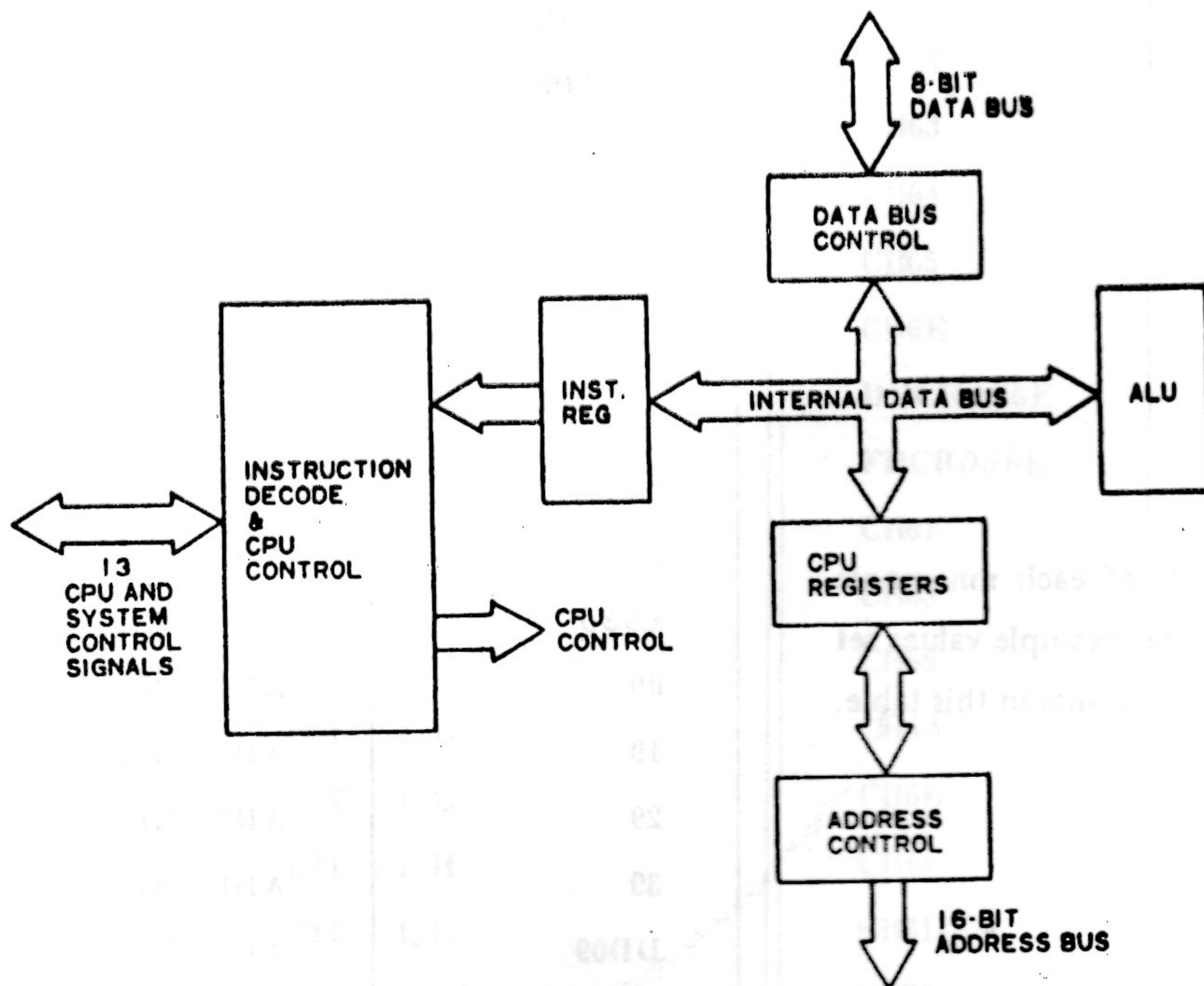
3D53

Cold start entry point

APPENDIX B

Z.80 OPERATION CODES.

ARCHITECTURE



Z-80 CPU BLOCK DIAGRAM

Mnemonic Codes and Corresponding Object Codes

(Mnemonic codes are arranged in alphabetic order.)

Note

nn, n, d and e in the operands of each mnemonic code represent constant data. The example values set forth below are used for these constants in this table.

nn = 584H

n = 20H

d = 5

e = 30H

Data codes represented by example values are shown in italic and underlined.

OP-Code	Mnemonic
8E	ADC A, (HL)
<u>DD8E05</u>	ADC A, (IX + d)
<u>FD8E05</u>	ADC A, (IY + d)
8F	ADC A, A
88	ADC A, B
89	ADC A, C
8A	ADC A, D
8B	ADC A, E
8C	ADC A, H
8D	ADC A, L
<u>CE20</u>	ADC A, n
ED4A	ADC HL, BC
ED5A	ADC HL, DE
ED6A	ADC HL, HL
ED7A	ADC HL, SP
86	ADD A, (HL)
<u>DD8605</u>	ADD A, (IX + d)
<u>FD8605</u>	ADD A, (IY + d)
87	ADD A, A
80	ADD A, B
81	ADD A, C
82	ADD A, D
83	ADD A, E
84	ADD A, H
85	ADD A, L
<u>C620</u>	ADD A, n
09	ADD HL, BC
19	ADD HL, DE
29	ADD HL, HL
39	ADD HL, SP
DD09	ADD IX, BC
DD19	ADD IX, DE
DD29	ADD IX, IX
DD39	ADD IX, SP
FD09	ADD IY, BC
FD19	ADD IY, DE
FD29	ADD IY, IY
FD39	ADD IY, SP

OP-Code	Mnemonic	OP-Code	Mnemonic
A6	AND (HL)	CB54	BIT 2, H
DDA <u>605</u>	AND (IX + d)	CB55	BIT 2, L
FDA <u>605</u>	AND (IY + d)	CB5E	BIT 3, (HL)
A7	AND A	DDCB <u>05 5E</u>	BIT 3, (IX + d)
A0	AND B	FDCB <u>05 5E</u>	BIT 3, (IY + d)
A1	AND C	CB5F	BIT 3, A
A2	AND D	CB58	BIT 3, B
A3	AND E	CB59	BIT 3, C
A4	AND H	CB5A	BIT 3, D
A5	AND L	CB5B	BIT 3, E
E6 <u>20</u>	AND n	CB5C	BIT 3, H
		CB5D	BIT 3, L
CB46	BIT 0, (HL)	CB66	BIT 4, (HL)
DDCB <u>05 46</u>	BIT 0, (IX + d)	DDCB <u>05 66</u>	BIT 4, (IX + d)
FDCB <u>05 46</u>	BIT 0, (IY + d)	FDCB <u>05 66</u>	BIT 4, (IY + d)
CB47	BIT 0, A	CB67	BIT 4, A
CB40	BIT 0, B	CB60	BIT 4, B
CB41	BIT 0, C	CB61	BIT 4, C
CB42	BIT 0, D	CB62	BIT 4, D
CB43	BIT 0, E	CB63	BIT 4, E
CB44	BIT 0, H	CB64	BIT 4, H
CB45	BIT 0, L	CB65	BIT 4, L
CB4E	BIT 1, (HL)	CB6E	BIT 5, (HL)
DDCB <u>05 4E</u>	BIT 1, (IX + d)	DDCB <u>05 6E</u>	BIT 5, (IX + d)
FDCB <u>05 4E</u>	BIT 1, (IY + d)	FDCB <u>05 6E</u>	BIT 5, (IY + d)
CB4F	BIT 1, A	CB6F	BIT 5, A
CB48	BIT 1, B	CB68	BIT 5, B
CB49	BIT 1, C	CB69	BIT 5, C
CB4A	BIT 1, D	CB6A	BIT 5, D
CB4B	BIT 1, E	CB6B	BIT 5, E
CB4C	BIT 1, H	CB6C	BIT 5, H
CB4D	BIT 1, L	CB6D	BIT 5, L
CB56	BIT 2, (HL)	CB76	BIT 6, (HL)
DDCB <u>05 56</u>	BIT 2, (IX + d)	DDCB <u>05 76</u>	BIT 6, (IX + d)
FDCB <u>05 56</u>	BIT 2, (IY + d)	FDCB <u>05 76</u>	BIT 6, (IY + d)
CB57	BIT 2, A	CB77	BIT 6, A
CB50	BIT 2, B	CB70	BIT 6, B
CB51	BIT 2, C	CB71	BIT 6, C
CB52	BIT 2, D	CB72	BIT 6, D
CB53	BIT 2, E	CB73	BIT 6, E

OP-Code	Mnemonic	OP-Code	Mnemonic
CB74	BIT 6, H	EDB1	CPIR
CB75	BIT 6, L	2F	CPL
CB7E	BIT 7, (HL)		
DDCB057E	BIT 7, (IX + d)	27	DAA
FDCB057E	BIT 7, (IY + d)		
CB7F	BIT 7, A	35 DD3505 FD3505 3D 05 0B 0D 15 1B 1D 25 2B DD2B FD2B 2D 3B	DEC (HL)
CB78	BIT 7, B		DEC (IX + d)
CB79	BIT 7, C		DEC (IY + d)
CB7A	BIT 7, D		DEC A
CB7B	BIT 7, E		DEC B
CB7C	BIT 7, H		DEC BC
CB7D	BIT 7, L		DEC C
DC8405 FC8405 D48405 CD8405 C48405 F48405 EC8405 E48405 CC8405	CALL C, nn		DEC D
	CALL M, nn		DEC DE
	CALL NC, nn		DEC E
	CALL nn		DEC H
	CALL NZ, nn		DEC HL
	CALL P, nn		DEC IX
	CALL PE, nn		DEC IY
	CALL PO, nn		DEC L
	CALL Z, nn		DEC SP
3F	CCF	F3	DI
BE DDBE05 FDBE05 BF B8 B9 BA BB BC BD FE20	CP (HL)	102E	DJNZ e
	CP (IX + d)	FB	EI
	CP (IY + d)		
	CP A	E3 DDE3 FDE3 08 EB D9	EX (SP), HL
	CP B		EX (SP), IX
	CP C		EX (SP), IY
	CP D		EX AF, AF'
	CP E		EX DE, HL
	CP H		EXX
	CP L	76	HALT
	CP n		
EDA9 EDB9 EDA1	CPD	ED46 ED56	IM 0
	CPDR		IM 1
	CPI		

OP-Code	Mnemonic	OP-Code	Mnemonic
ED5E	IM 2	C28405	JP NZ,nn
ED78	IN A,(C)	F28405	JP P,nn
DB20	IN A,(n)	EA8405	JP PE,nn
ED40	IN B,(C)	F28405	JP PO,nn
ED48	IN C,(C)	CA8405	JP Z,nn
ED50	IN D,(C)	382E	JR C,e
ED58	IN E,(C)	182E	JR e
ED60	IN H,(C)	302E	JR NC,e
ED68	IN L,(C)	202E	JR NZ,e
34	INC (HL)	282E	JR Z,e
DD3405	INC (IX+d)	02	LD (BC),A
FD3405	INC (IY+d)	12	LD (DE),A
3C	INC A	77	LD (HL),A
04	INC B	70	LD (HL),B
03	INC BC	71	LD (HL),C
0C	INC C	72	LD (HL),D
14	INC D	73	LD (HL),E
13	INC DE	74	LD (HL),H
1C	INC E	75	LD (HL),L
24	INC H	3620	LD (HL),n
23	INC HL	DD7705	LD (IX+d),A
DD23	INC IX	DD7005	LD (IX+d),B
FD23	INC IY	DD7105	LD (IX+d),C
2C	INC L	DD7205	LD (IX+d),D
33	INC SP	DD7305	LD (IX+d),E
EDAA	IND	DD7405	LD (IX+d),H
EDBA	INDR	DD7505	LD (IX+d),L
EDA2	INI	DD360520	LD (IX+d),n
EDB2	INIR	FD7705	LD (IY+d),A
E9	JP (HL)	FD7005	LD (IY+d),B
DDE9	JP (IX)	FD7105	LD (IY+d),C
FDE9	JP (IY)	FD7205	LD (IY+d),D
DA8405	JP C,nn	FD7305	LD (IY+d),E
FA8405	JP M,nn	FD7405	LD (IY+d),H
D28405	JP NC,nn	FD7505	LD (IY+d),L
C38405	JP nn	FD360520	LD (IY+d),n
		328405	LD (nn),A
		ED438405	LD (nn),BC

OP-Code	Mnemonic	OP-Code	Mnemonic
<u>ED538405</u>	LD (nn), DE	4B	LD C, E
<u>228405</u>	LD (nn), HL	4C	LD C, H
<u>DD228405</u>	LD (nn), IX	4D	LD C, L
<u>FD228405</u>	LD (nn), IY	<u>0E20</u>	LD C, n
<u>ED738405</u>	LD (nn), SP	56	LD D, (HL)
0A	LD A, (BC)	<u>DD5605</u>	LD D, (IX + d)
1A	LD A, (DE)	<u>FD5605</u>	LD D, (IY + d)
7E	LD A, (HL)	57	LD D, A
<u>DD7E05</u>	LD A, (IX + d)	50	LD D, B
<u>FD7E05</u>	LD A, (IY + d)	51	LD D, C
<u>3A8405</u>	LD A, (nn)	52	LD D, D
7F	LD A, A	53	LD D, E
78	LD A, B	54	LD D, H
79	LD A, C	55	LD D, L
7A	LD A, D	<u>1620</u>	LD D, n
7B	LD A, E	<u>ED5B8405</u>	LD DE, (nn)
7C	LD A, H	<u>118405</u>	LD DE, nn
ED57	LD A, I	5E	LD E, (HL)
7D	LD A, L	<u>DD5E05</u>	LD E, (IX + d)
<u>3E20</u>	LD A, n	<u>FD5E05</u>	LD E, (IY + d)
46	LD B, (HL)	5F	LD E, A
<u>DD4605</u>	LD B, (IX + d)	58	LD E, B
<u>FD4605</u>	LD B, (IY + d)	59	LD E, C
47	LD B, A	5A	LD E, D
40	LD B, B	5B	LD E, E
41	LD B, C	5C	LD E, H
42	LD B, D	5D	LD E, L
43	LD B, E	<u>1E20</u>	LD E, n
44	LD B, H	66	LD H, (HL)
45	LD B, L	<u>DD6605</u>	LD H, (IX + d)
<u>0620</u>	LD B, n	<u>FD6605</u>	LD H, (IY + d)
<u>ED4B8405</u>	LD BC, (nn)	67	LD H, A
<u>018405</u>	LD BC, nn	60	LD H, B
4E	LD C, (HL)	61	LD H, C
<u>DD4E05</u>	LD C, (IX + d)	62	LD H, D
<u>FD4E05</u>	LD C, (IY + d)	63	LD H, E
4F	LD C, A	64	LD H, H
48	LD C, B	65	LD H, L
49	LD C, C	<u>2620</u>	LD H, n
4A	LD C, D	<u>2A8405</u>	LD H, (nn)

OP-Code	Mnemonic	OP-Code	Mnemonic
<u>218405</u>	LD HL,nn	B4	OR H
ED47	LD I,A	B5	OR L
DD2A <u>8405</u>	LD IX,(nn)	<u>F620</u>	OR n
DD21 <u>8405</u>	LD IX,nn	EDBB	OTDR
FD2A <u>8405</u>	LD IY,(nn)	EDB3	OTIR
FD21 <u>8405</u>	LD IY,nn	ED79	OUT (C),A
6E	LD L,(HL)	ED41	OUT (C),B
DD6E <u>05</u>	LD L,(IX+d)	ED49	OUT (C),C
FD6E <u>05</u>	LD L,(IY+d)	ED51	OUT (C),D
6F	LD L,A	ED59	OUT (C),E
68	LD L,B	ED61	OUT (C),H
69	LD L,C	ED69	OUT (C),L
6A	LD L,D	<u>D320</u>	OUT (n),A
6B	LD L,E	EDAB	OUTD
6C	LD L,H	EDA3	OUTI
6D	LD L,L		
<u>2E20</u>	LD L,n	F1	POP AF
ED7R <u>8405</u>	LD SP,(nn)	C1	POP BC
F9	LD SP,HL	D1	POP DE
DDF9	LD SP,IX	E1	POP HL
FDF9	LD SP,IY	DDE1	POP IX
<u>318405</u>	LD SP,nn	FDE1	POP IY
EDA8	LDD	F5	PUSH AF
EDB8	LDDR	C5	PUSH BC
EDA0	LDI	D5	PUSH DE
EDB0	LDIR	E5	PUSH HL
ED44	NEG	DDE5	PUSH IX
00	NOP	FDE5	PUSH IY
B6	OR (HL)	CB86	RES 0,(HL)
DDB6 <u>05</u>	OR (IX+d)	DDCB <u>0586</u>	RES 0,(IX+d)
FDB6 <u>05</u>	OR (IY+d)	FDCB <u>0586</u>	RES 0,(IY+d)
B7	OR A	CB87	RES 0,A
B0	OR B	CB80	RES 0,B
B1	OR C	CB81	RES 0,C
B2	OR D	CB82	RES 0,D
B3	OR E	CB83	RES 0,E
		CB84	RES 0,H

OP-Code	Mnemonic	OP-Code	Mnemonic
CB85	RES 0, L	CBA5	RES 4, L
CB8E	RES 1, (HL)	CBAE	RES 5, (HL)
DDCB058E	RES 1, (IX+d)	DDCB05AE	RES 5, (IX+d)
FDCB058E	RES 1, (IY+d)	FDCB05AE	RES 5, (IY+d)
CB8F	RES 1, A	CBAF	RES 5, A
CB88	RES 1, B	CBA8	RES 5, B
CB89	RES 1, C	CBA9	RES 5, C
CB8A	RES 1, D	CBA A	RES 5, D
CB8B	RES 1, E	CBAB	RES 5, E
CB8C	RES 1, H	CBAC	RES 5, H
CB8D	RES 1, L	CBAD	RES 5, L
CB96	RES 2, (HL)	CBB6	RES 6, (HL)
DDCB0596	RES 2, (IX+d)	DDCB05B6	RES 6, (IX+d)
FDCB0596	RES 2, (IY+d)	FDCB05B6	RES 6, (IY+d)
CB97	RES 2, A	CBB7	RES 6, A
CB90	RES 2, B	CBB0	RES 6, B
CB91	RES 2, C	CBB1	RES 6, C
CB92	RES 2, D	CBB2	RES 6, D
CB93	RES 2, E	CBB3	RES 6, E
CB94	RES 2, H	CBB4	RES 6, H
CB95	RES 2, L	CBB5	RES 6, L
CB9E	RES 3, (HL)	CBBE	RES 7, (HL)
DDCB059E	RES 3, (IX+d)	DDCB05BE	RES 7, (IX+d)
FDCB059E	RES 3, (IY+d)	FDCB05BE	RES 7, (IY+d)
CB9F	RES 3, A	CBBF	RES 7, A
CB98	RES 3, B	CBB8	RES 7, B
CB99	RES 3, C	CBB9	RES 7, C
CB9A	RES 3, D	CBBA	RES 7, D
CB9B	RES 3, E	CBBB	RES 7, E
CB9C	RES 3, H	CBBC	RES 7, H
CB9D	RES 3, L	CBB D	RES 7, L
CBA6	RES 4, (HL)		
DDCB05A6	RES 4, (IX+d)	C9	RET
FDCB05A6	RES 4, (IY+d)	D8	RET C
CBA7	RES 4, A	F8	RET M
CBA0	RES 4, B	D0	RET NC
CBA1	RES 4, C	C0	RET NZ
CBA2	RES 4, D	F0	RET P
CBA3	RES 4, E	E8	RET PE
CBA4	RES 4, H	E0	RET PO

OP-Code	Mnemonic	OP-Code	Mnemonic
C8	RET Z	CB0E	RRC (HL)
ED4D	RETI	DDCB050E	RRC (IX+d)
ED45	RETN	FDCB050E	RRC (IY+d)
CB16	RL (HL)	CB0F	RRC A
DDCB0516	RL (IX+d)	CB08	RRC B
FDCB0516	RL (IY+d)	CB09	RRC C
CB17	RL A	CB0A	RRC D
CB10	RL B	CB0B	RRC E
CB11	RL C	CB0C	RRC H
CB12	RL D	CB0D	RRC L
CB13	RL E	0F	RRCA
CB14	RL H	ED67	RRD
CB15	RL L	C7	RST 00H
17	RLA	CF	RST 08H
CB06	RLC (HL)	D7	RST 10H
DDCB0506	RLC (IX+d)	DF	RST 18H
FDCB0506	RLC (IY+d)	E7	RST 20H
CB07	RLC A	EF	RST 28H
CB00	RLC B	F7	RST 30H
CB01	RLC C	FF	RST 38H
CB02	RLC D	9E	SBC A,(HL)
CB03	RLC E	DD9E05	SBC A,(IX+d)
CB04	RLC H	FD9E05	SBC A,(IY+d)
CB05	RLC L	9F	SBC A,A
07	RLCA	98	SBC A,B
ED6F	RLD	99	SBC A,C
CB1E	RR (HL)	9A	SBC A,D
DDCB051E	RR (IX+d)	9B	SBC A,E
FDCB051E	RR (IY+d)	9C	SBC A,H
CB1F	RR A	9D	SBC A,L
CB18	RR B	DE20	SBC A,n
CB19	RR C	ED42	SBC HL,BC
CB1A	RR D	ED52	SBC HL,DE
CB1B	RR E	ED62	SBC HL,HL
CB1C	RR H	ED72	SBC HL,SP
CB1D	RR L	37	SCF
1F	RRA		

OP-Code	Mnemonic	OP-Code	Mnemonic
CBC6	SET 0, (HL)	CBE6	SET 4, (HL)
DDCB05C6	SET 0, (IX + d)	DDCB05 E6	SET 4, (IX + d)
FDCB05C6	SET 0, (IY + d)	FDCB05 E6	SET 4, (IY + d)
CBC7	SET 0, A	CBE7	SET 4, A
CBC0	SET 0, B	CBE0	SET 4, B
CBC1	SET 0, C	CBE1	SET 4, C
CBC2	SET 0, D	CBE2	SET 4, D
CBC3	SET 0, E	CBE3	SET 4, E
CBC4	SET 0, H	CBE4	SET 4, H
CBC5	SET 0, L	CBE5	SET 4, L
CBCE	SET 1, (HL)	CBEE	SET 5, (HL)
DDCB05CE	SET 1, (IX + d)	DDCB05 EE	SET 5, (IX + d)
FDCB05CE	SET 1, (IY + d)	FDCB05 EE	SET 5, (IY + d)
CBCF	SET 1, A	CBEF	SET 5, A
CBC8	SET 1, B	CBE8	SET 5, B
CBC9	SET 1, C	CBE9	SET 5, C
CBCA	SET 1, D	CBEA	SET 5, D
CBCB	SET 1, E	CBEB	SET 5, E
CBCC	SET 1, H	CBEC	SET 5, H
CBCD	SET 1, L	CBED	SET 5, L
CBD6	SET 2, (HL)	CBF6	SET 6, (HL)
DDCB05 D6	SET 2, (IX + d)	DDCB05 F6	SET 6, (IX + d)
FDCB05 D6	SET 2, (IY + d)	FDCB05 F6	SET 6, (IY + d)
CBD7	SET 2, A	CBF7	SET 6, A
CBD0	SET 2, B	CBF0	SET 6, B
CBD1	SET 2, C	CBF1	SET 6, C
CBD2	SET 2, D	CBF2	SET 6, D
CBD3	SET 2, E	CBF3	SET 6, E
CBD4	SET 2, H	CBF4	SET 6, H
CBD5	SET 2, L	CBF5	SET 6, L
CBD8	SET 3, B	CBFE	SET 7, (HL)
CBDE	SET 3, (HL)	DDCB05 FE	SET 7, (IX + d)
DDCB05 DE	SET 3, (IX + d)	FDCB05 FE	SET 7, (IY + d)
FDCB05 DE	SET 3, (IY + d)	CBFF	SET 7, A
CBDF	SET 3, A	CBF8	SET 7, B
CBD9	SET 3, C	CBF9	SET 7, C
CBDA	SET 3, D	CBFA	SET 7, D
CBDB	SET 3, E	CBFB	SET 7, E
CBDC	SET 3, H	CBFC	SET 7, H
CBDD	SET 3, L	CBFD	SET 7, L

OP-Code	Mnemonic	OP-Code	Mnemonic
CB26	SLA (HL)	93	SUB E
DDCB05 26	SLA (IX + d)	94	SUB H
FDCB05 26	SLA (IY + d)	95	SUB L
CB27	SLA A	D620	SUB n
CB20	SLA B		
CB21	SLA C	AE	XOR (HL)
CB22	SLA D	DDAE05	XOR (IX + d)
CB23	SLA E	FDAE05	XOR (IY + d)
CB24	SLA H	AF	XOR A
CB25	SLA L	A8	XOR B
		A9	XOR C
CB2E	SRA (HL)	AA	XOR D
DDCB05 2E	SRA (IX + d)	AB	XOR E
FDCB05 2E	SRA (IY + d)	AC	XOR H
CB2F	SRA A	AD	XOR L
CB28	SRA B	EE20	XOR n
CB29	SRA C		
CB2A	SRA D		
CB2B	SRA E		
CB2C	SRA H		
CB2D	SRA L		
CB3E	SRL (HL)		
DDCB05 3E	SRL (IX + d)		
FDCB05 3E	SRL (IY + d)		
CB3F	SRL A		
CB38	SRL B		
CB39	SRL C		
CB3A	SRL D		
CB3B	SRL E		
CB3C	SRL H		
CB3D	SRL L		
96	SUB (HL)		
DD9605	SUB (IX + d)		
FD9605	SUB (IY + d)		
97	SUB A		
90	SUB B		
91	SUB C		
92	SUB D		

INPUT / OUTPUT PORTS

All addresses are in hexadecimal.

	Port	i/o	Function
<u>PSG</u>			
	02	write	register data data
	03	write	
<u>VDP</u>			
	08	Read/write	Send Vram data
	09	write only	Send register data
	09	read only	Read Vdp status register
<u>PCI</u>			
	10	read/write	Data register
	11	read/write	Control/status register
<u>FDC</u>			
	18	read/write	Status/command register
	19	read/write	Track register
	1A	read/write	Sector register
	1B	read/write	Data register
<u>ROM SELECT</u>			
	24	read/write	Toggles Mos/memory
<u>FIRE button</u>			

THE SOURCE APPENDIX C

25 write only

Bit 0 = 1 to mask
bits 1 - 7 not used

AUXILIARY COMMAND status register

Port	i/o	Function
20	read only	Bits 0 = Fire 1 1 = Fire 2 2 = printer busy 3 = Paper empty 4 = printer error 5 = graphic/alpha key 6 = Control key 7 = Shift key Bit 0 = keyboard int mask '1' = mask other bits n/a
	write only	

QUE/DIGITAL MASK

21 write only

Bit 0 = 1 to mask
other bits n/a

22 read/write

Alpha lock LED on/off
each port access.

DRIVE SELECT

23 write only

Bits 0 = drive 1
 1 = drive 2
 2 = drive 3
 3 = drive 4
set to 1 to select drive
 4 = side select

CTC

28	read/write	Channel 0
29	read/write	Channel 1
2A	read/write	Channel 2
2B	read/write	Channel 3

MJS CALLS

- 80 AR TH Performs as '(ARITHMETIC) from MOS
- Values Passed
 xxxx in HL pair
 yyyy in DE pair (See Manual)
- 81 BAL D Performs as 'B' (BAUD) from MOS
- Values Passed
 x - Receive rate - upper nibble of L register
 y - Transmit rate - lower nibble of L register
 ww - Mode Byte - D register
 zz - Command Byte - E register (see manual)
- If DE is zero, the mode and command bytes will remain unchanged.
 For the correct receive and transmit rates, the baud rate X16 must be used.
- 82 COFY Performs as 'C' (COPY) from MOS
- Values Passed
 xxxx - Start - in HL pair
 yyyy - finish - in DE pair
 zzzz - destination - in BC pair (see manual)
- 83 DECIML Performs as 'D' (DECIMAL) from MOS
- Values Passed
 xxxx in HL pair (see manual)
- 84 EXEC Performs as 'E' (EXECUTE) from MOS

		<p>Values Passed xxxx - Break point - in HL pair (see manual)</p>
85	MFILL	<p>Performs as 'F' (FILL) from MOS</p>
		<p>Values Passed xxxx - Start - HL pair yyyy - Finish - DE pair zz - Value - C register (see manual)</p>
86	GOTO	<p>Performs as 'G' (GOTO) from MOS</p>
		<p>Values Passed xxxx - execution address - HL pair yyyy - break point - DE pair (see manual)</p>
		<p>If DE is zero, no break point is set</p>
87	HEX	<p>Performs as 'H' (HEXADECIMAL) from MOS</p>
		<p>Values Passed Pointer to text (in RAM) - DE pair</p> <p>The decimal number is held at (DE) and terminated with 00 The hexadecimal number is returned in the HL pair in addition to being displayed.</p>
8C	MODIFY	<p>Performs as 'M' (MODIFY) from MOS</p>
		<p>Values Passed xxxx - address to modify from - HL pair</p>
91	RDBLOK	<p>Performs as 'R' (READ) from MOS</p>
		<p>Values Passed Pointer to text (in RAM) - DE pair The text takes the format as</p>

shown in the manual.
(See ZRBLK function no.A4)

95

TBLATE

Performs as 'T' (TABULATE) from MOS

Values Passed

xxxx - Start address - HL pair

yyyy - finish address - DE pair

zz - no. of columns - C register

If C is passed as zero, zz will default to 8 columns

96

WRBLOK

Performs as 'W' (WRITE) from MOS

Values Passed

Pointer to text (in RAM) - DE pair

The text at (DE) takes the format as shown in the manual
(see ZWBLK function no. A5)

97

COLD

Performs as 'X' (COLD START) from MOS

98

WARM

Performs as 'Y' (WARM START) from MOS

99

REGSTR

Performs as 'Z' (REGISTER EXAMINE) from MOS

Values Passed

x - registers to be displayed 1 register (0, 1 or 2)

Note: This does not display the current register contents but the register contents at the last RST 38H (FFH encountered)

9A

ZINIT

Re-entry point to MOS

9B

ZRSCAN

- Repeat key scan

This will return the value of any

THE SOURCE APPENDIX D

key pressed, in the A register.
00 is returned if no key is pressed.

00 can also be returned if the keyboard poll rate (polled with this MCAL) is greater than the key repeat speed. This repeat speed can be altered in scratch pad location FB43H

Note: This works as the KBD command in XBAS

9C ZKEYIN

- Input key

This will return the value of any key pressed, in A register.

Unlike MCAL 9B, this will wait for a key to be pressed.

Note: This is similar to the INCH command in XBAS

9D ZGETLN

- Get Text from keyboard

This will enter a line of text from the keyboard into RAM from the address held in the DE pair. The text is displayed on the screen on pressing the keys and the line is terminated with an ENTER

9E ZOUTC

- Character Output

Outputs a character to screen held in the A register.

9F ZPOUT

Outputs a character to the parallel printer held in the A register.

A0 ZSLOUT

- Serial Output

Outputs a character to the serial port from the A register.

A1 ZSRLIN

- Serial Input

A2

ZRSECT

Read a byte from the serial port and returns the value in the A register.

- Sector Read

Reads a sector from the disc into the sector buffer (A sector is 200H bytes)

The following are set up in the scratch pad

Location Label

FB50H HSTDSC - Disc drive (0-3)
 FB51H HSTTRK - Track (0-27H)
 FB52H HSTSEC - Sector (0-9)
 FB53H HSTDMA - Sector buffer address (normally FE00H)

A3

ZWSECT

- Sector Write

Writes a sector from the disc into the sector buffer (see FB50H A2)

A4

ZRBLK

Read Block

Read a block of data from the disc

Values Passed

Drive no. (0-3) in A register
 Start address in HL pair
 Finish address in DE pair
 Sector (0-9) in B register
 Track (0-27H) in C register
 Memory is filled to the next complete sector (220H bytes) ie. if the start and finish address is specified as 6000H and 6001H respectively, 6000H to 6200H will be read from the disc

A5

ZWBLK

Write Block

Writes a block of data to the disc.

The values passed are the same as

THE SOURCE APPENDIX D

A6	ZCRLF	for MCAL A4 and again is written to the next complete sector (200H bytes)
A7	ZCRLFZ	Outputs a CR (ODH) and LF (OAH)
A8	ZSPACE	Outputs a CR and LF if the cursor is not at column zero.
A9	ZPR4HX	Outputs 1 space
AA	ZP2HXZ	Outputs 4 hex digits held in the HL pair. eg = 1234H, 1234 is output
AB	ZPR2HX	Outputs two hex digits held in the A register followed by a space
AC	ZFC4HX	Outputs two hex digits held in the A register (as MCAL AA with no space output)
AD	ZFCZHX	Get a hex number (up to 4 digits) from text into the HL pair. DE points to the text (in RAM). The number is terminated on a non hex character.
AE	ZDCHD	Get a hex number (up to 2 digits) from text into the A register. DE points to the text. The number is terminated with a non hex character.
AF	ZHNDSC	Outputs a command to the FDC. The command type is passed in the A register. On return A is set to 00 unless the FDC is not executing the command then A is set to FFH.
		Takes the drive head to track 00

The drive no. is passed in the A register

B0	ZIGBLK	Returns a value in the A register from RAM pointed to by the DE pair if DE > 7FFFH, or from ROM if DE < 8000H. Note: Commas and spaces are ignored (ie the first non comma/non zero character is displayed)
B1	ZRDMEM	Returns a value in the A register from RAM pointed to by the HL pair.
B2	ZRCPYU	Performs an LDIR instruction (switches ROM out first)
B3	ZRCPYD	Performs an LDDR instruction (switches ROM out first)
B4	ZMOUT	Outputs a value held in the B register to the PSG port no. held in the C register.
B5	ZKSCAN	Returns the value in the A register of any key pressed. 00 is returned if no key is pressed. This is similar to MCAL 9B (ZRSCAN) except that it is unaffected by the key repeat speed. That is for each MCAL execution, a value is returned.
BC	ZZTIME	Set up CTC channels 2 and 3 to generate 1 second interrupts for clock
BD	ZFDRST	Resets the FDC after an error. Also resets the PSG (using MCAL C0 (ZPINIT))

B0 **ZIGBLK**

The drive no. is passed in the A register

Returns a value in the A register from RAM pointed to by the DE pair if DE > 7FFFH, or from ROM if DE < 8000H.

Note: Commas and spaces are ignored (ie the first non comma/non zero character is displayed)

B1 **ZRDMEM**

Returns a value in the A register from RAM pointed to by the HL pair.

B2 **ZRCPYU**

Performs an LDIR instruction (switches ROM out first)

B3 **ZRCPYD**

Performs an LDDR instruction (switches ROM out first)

B4 **ZMOUT**

Outputs a value held in the B register to the PSG port no. held in the C register.

B5 **ZKSCAN**

Returns the value in the A register of any key pressed. 00 is returned if no key is pressed.

This is similar to MCAL 9B (ZRSCAN) except that it is unaffected by the key repeat speed. That is for each MCAL execution, a value is returned.

BC **ZZTIME**

Set up CTC channels 2 and 3 to generate 1 second interrupts for clock

BD **ZFDRST**

Resets the FDC after an error. Also resets the PSG (using MCAL CO (ZPINIT))

THE SOURCE APPENDIX D

BE ZSYSRS

This performs the following:-

1. Clears the screen to 40 columns
2. Resets all characters
3. Removes sprites
4. Resets the FDC and PSG
5. Masks the keyboard, fire and ADC interrupts.

BF ZLOGO

Outputs ***EINSTEIN*** Logo

C0 ZPINIT

Sets PSG register 7 to 7FH and all other registers to 00

C1 ZSREG

Sends an address held in the BC pair to the VDP. Data can then be output to VRAM from port 8. Subsequent data bytes sent will be loaded into subsequent VRAM locations.

Note: A delay of 8H is necessary between any VRAM reads or writes (eg PUSH/POP)

Note: When ROM is switched in, RST 20H will execute this MCAL.

C2 ZVRIN

Returns a value in the A register from the VRAM address pointed to by the BC pair

C3 ZVROUT

Writes data held in the A register to the VRAM address held in the BC pair.

CE ZIMULT

Multiplies the contents of the DE pair and the contents of the BC pair and returns the value in DEHL (The DE pair is the most significant)

CF ZPRM

Outputs a message to the screen. The data follows the CF data byte and must be a character in the range 0 to 7FH. The message is terminated by adding 80H to the last character in the message.

D0 ZVOUT

Outputs a character from the A register to the current cursor position without incrementing the cursor position. This can be useful to prevent scrolling, linefeeds etc.

D1 ZSCURS

Returns VRAM addresses relating to the current cursor position. The ASCII text map address is returned in the BC pair (will be in the range 3C00H to 3FBFH). The table pattern generator address (first byte) is returned in the DE pair (will be in the range 0000 to 17FFH). The start of the sprite pattern/text pattern table is returned in the HL pair (normally 1800H).

D2 ZROM

Starts execution from address 4004H in the 2nd ROM. All registers can be used to pass and return values. Return is achieved by a RET instruction.

2nd ROM protocol

Execution from here if 4000H is zero on power up or reset after printing Einstein logo and before booting any disc present. A RET will return execution to the disc auto boot routine.

4000H 4001H 4002H 4003H 4004H

ROM detection	Execution from
byte	here on MCAL D2
00 - for auto	
execution	

C4 ZPLOT

PLOTS or UNPLOTS a pixel

A = 1 for PLOT

C5

ZPL TXY

A + 0 for UNPLOT
IX holds the x coordinate
IY hold the y coordinate

Plots a point according to the line type (see below)

```
IX holds the x coordinate
IY holds the y coordinate
```

Line Type

Four scratch pad values contain information as to the line type to be drawn e.g.

these are:-

SCRATCH LOCATION

FBA8H DOTON - Length to end of
first line

FBA9H DOTOFF - Length to end of
first space

```
FBAAH DOTON2 - Length to end of
Returns a value second line
```

FBABH DOTOF2 - Length to end of
by the DC part: second space

```
<- line
type
```

DOTON ter to

DOTOFF

DOTON2

DOTOF 3

Normally these 4 values are in ascending order. For a continuous line, DOTON is set to FFH and the other 3 zero.

For a continuous unplot, DOTOFF
is set to FFH and the other 3
zero

Note: If all line type bytes are zero, the system will hang up.

C6 ZPOINT

Returns the status of any pixel in the Z register.

1 = foreground
0 = background

BC contains the x coordinate
DE contains the y coordinate

The VRAM address of the pixel is returned in the BC pair.

C7 ZPNTXY

Returns the status of any pixel in the A register.

This is identical to MCAL C6 except IX contains the x coordinate and IY contains the y coordinate.

The VRAM address of the pixel is returned in the BC pair.

C8 ZDRWTO

This will draw a line from the coordinates held in the IX and IY registers to values in scratch pad locations FB96H (Xi) and FB98H (Y1). Each is a two byte number. The type of line drawn is determined by the line type values (see MCAL C5)

C9 ZPOLYG

This draws a polygon (or ellipse)

The polygon centre coordinates are held in scratch pad locations FB9EG (CX) and FBA0H (CY), each is a two byte number.

The horizontal and vertical radii are held in locations FBA2H (RADX-2BYTE) and FBA4H (RADY-2BYTE).

The number of sides on the polygon is determined by a two byte number in scratch pad location FBA6H (CINC).

A value of 4 will give a circle.
A value of 80H will give an

octagon.

A value of 100H will give a rectangle etc.

The start angle is passed in the DE pair, the finish angle is passed in the BC pair, each is in the range 0 to 1024.

The lines drawn to the polygon centre are selected by setting the carry flag.

The type of line drawn is determined by the line type values (see MCAL C5)

Adds the x coordinates Origin value held in scratch location FB9AH (ORGX - 2Byte) to the contents of BC and returns the result in the BC pair and adds the y coordinate Origin value held in scratch location FB9CH (ORGY - 2Byte) to the contents of DE and returns the result in the DE pair.

This MCAL is of little use and is called from within other graphics MCALS.

The values ORGX and ORGY are normally zero but when altered will cause all graphics output to be offset by that value. This is similar to the ORIGIN command in XBAS.

Returns the VRAM address in the BC pair for coordinates x and y passed in the IX and IY registers respectively. The pixel position within the 8 pixel row is returned in the E register, counting from the left hand pixel (0-7)

Writes data held in the A register to the pattern generator table address passed in the BC pair (0 to 17FFH) and sets the corresponding byte in the pattern

CA ZORGC0

CB XCALAD

CC ZSETCL

colour table (2000H to 37FFH) to the contents of scratch locations FB39H (GCOLR)

CD

ZFILL

Fills an area on screen surrounding coordinates passed in the IX and IY registers (x and y coordinates respectively).

If the fill is in foreground (i.e. the point at x, y is not set) then scratch location FBADH (FILLMOD) must be set to FFH.

If the fill is in background (i.e. the point at x, y is set) then scratch location FBADH (FILLMOD) must be set to zero.

MCAL XPNTXY (C7) can be used to find the fill type needed.

Protocol:

Output a character to screen

```
LD    A, 'x'
RST   8
DB    9E
```


APPENDIX E

THE SUBROUTINES

;
 ;Random number routine. Call this routine then load parameters into PARA which is
 ;a two byte word. Result is return in VAL BC must be preserved until
 ;result has been obtained. Procedure: LD A,R:CALL
 ;RND: LD HL,PARAMS: LD A,HIGH
 ;VALUE: CALL PARA : LD A,(VAL) = RANDOM NUMBER
 ;

RND:

```

PUSH    AF
PUSH    BC
PUSH    DE
PUSH    HL
LD      A,R
LD      (SEED3),A
LD      DE,(SEED)
LD      HL,(SEED2)
LD      B,07

```

RND10:

```

CALL    SHIFT
DJNZ    RND10
LD      B,03

```

RND20:

```

CALL    SUB
DJNZ    RND20
LD      (SEED),DE
LD      (SEED2),HL
LD      A,7FH
AND     D
LD      (VAL),A
POP     HL
POP     DE
POP     BC
POP     AF
RET

```

SHIFT:

```

ADD     HL,HL
EX      DE,HL
ADC     HL,HL
EX      DE,HL
RET

```

;TEMP STORE FOR RANDOM NUMBER BEFORE
 ;CALLING PARAMS


```

SUB:
    PUSH    BC
    LD      BC, (SEED2)
    OR      A
    SBC     HL, BC
    EX      DE, HL
    LD      BC, (SEED)
    SBC     HL, BC
    EX      DE, HL
    POP     BC
    RET

SEED:  DB    00H
SEED1: DB    00H
SEED2: DB    00H
SEED3: DB    00H
VAL:   DB    00H, 00H
PARA:  DB    00H, 00H
;
PARAMS:
    PUSH    BC
    PUSH    DE
    PUSH    HL
    LD      DE, (PARA)
    LD      A, (VAL)
    LD      L, A
    LD      H, 0

PARA1:
    LD      A, L
    CP      E
    JR      Z, PARA2
    JR      C, PARA2
    SBC     HL, DE
    JR      PARA1

PARA2:
    LD      A, L
    CP      00H
    JR      NZ, PARA3
    INC     HL

PARA3:
    LD      (VAL), HL
    POP     HL
    POP     DE
    POP     BC
    RET

```



```

;ROUTINE TO LOAD CHARACTER INTO VRAM AND SET COLOUR BYTE
;
;New routine to load character into Vram and set the colour byte
;Routine to be called from main loop.
;
;*****
TEXT EQU 1900H
;*****
;
;Assumes that IX register set up as follows:::
;IX+00H points to X co-ordinate
;IX+01H points to Y co-ordinate
;IX+02H holds current character value
;IX+03H holds current ink colour value
;IX+04H holds paper colour
;Preserves HL,BC,DE registers
;
;*****
;
KSUB1: PUSH BC
      PUSH DE
      PUSH HL
      LD IX,XCORD ;ALIGN IX FOR ABOVE TABLE
      LD (IX+02H),A ;SAVE CURRENT CHARACTER
      CP 32 ;IS IT A CONTROL CHARACTER?
      JP C,CONTROL ;LESS THAN 32 YES IT IS A CONTROL.
      SUB 32 ;GET RID OF ALL CONTROL CODES
      LD D,0 ;WE CAN NOW ALIGN TO RIGHT POSITION
      LD E,A ;DO ALIGNMENT
      SLA E ;
      RL D ;MAKE SURE NO FALLOUT FROM E ESCAPES!
      SLA E
      RL D
      SLA E
      RL D ;NOW CHAR*8 = CORRECT POSITION.
      LD HL,TEXT ;POINT TO START OF TEXT IN VRAM = EQU***
      ADD HL,DE ;ALIGNED TO CORRECT CHARACTER
      EX DE,HL ;DE=CHARACTER IN VRAM
      PUSH DE ;WE WILL NEED IT AGAIN SO SAVE IT
;

```

.COMMENT ?

The following routine converts the X,Y co-ordinates to Vram position which in the case of a Bit Mapped screen is set up from pattern generator table located at 0000H through to 17FFH in Vram.

Formula for calculations is based on 32 characters per line & 8 bytes per character.

Position in Vram = No lines in Ycord * 32 * 8 + (Number of chars in Xcord*8)


```

?
LD      H,0
LD      L,(IX+01H)    ;Y CO-ORDINATE
LD      B,08          ;FOR * 256 TRICK

LPX:
ADD     HL,HL          ;*2 *4 *8 ETC....
DJNZ    LPX
LD      D,0           ;HL=Y*32 * 8 = NO COLS PER LINE *
                        ;NUMBER OF BYTES PER CHARACTER
LD      E,(IX+00H)    ;GET X CO-ORD
SLA     E
SLA     E              ;*8 FOR NUMBER OF BYTES PER CHARACTER
SLA     E
ADD     HL,DE          ;HL-> TO CORRECT PLACE IN VRAM
POP     DE             ;DE STILL -> TO CHARACTER IN VRAM
CALL    SENDX         ;GO AND SEND IT
;
;
LD      A,(IX+00H)    ;X CO-ORD
INC     A              ;INCREMENT X CO-ORD
LD      (IX+00H),A
CP      32
JR      NZ,BACK       ;IF NOT MAX GO BACK
XOR     A              ;ZERO A REG MAX SCREEN WIDTH REACHED
LD      (IX+00H),A    ;RESET X CO-ORD

LNFED:
LD      A,(IX+01H)    ;GET Y CO-ORD
INC     A
CP      24             ;MAX DEPTH ?
JR      NZ,BACK
XOR     A
LD      (IX+01H),A

;
BACK:
POP     HL
POP     DE
POP     BC
RET

```

.COMMENT *

HL NOW -> TO VRAM POSITION FOR BIT-MAPPED MODE WHICH IS IN FACT THE PATTERN GENERATOR TABLE STARTING AT 0000H IN VRAM.

*

SENDX:

PUSH HL ;SAVE PRINT POSITION

;

;

.COMMENT *

The following routine takes the screen position pointed to by HL
the character bytes (in Vram) pointed to by DE.

Reads them from Vram and prints them at the screen position held in HL. It then sets the CORRECT COLOUR BYTES from the location pointed to by (IX+03H)

```

*
;
LD      B,08H
LPA:    EX      DE,HL      ;HL= PATTERN ADDRESS:DE = SCREEN POS
        CALL    ADDIN      ;SEND DATA
        IN      A,(VRM)    ;GET BYTE OF CHARACTER
        EX      DE,HL      ;AS WE WERE!
        CALL    ADDOUT     ;NOW SEND SCREEN ADDRESS
        OUT     (VRM),A    ;SEND DATA
        INC     HL         ;INC SCREEN
        INC     DE         ;INC CHARACTER POINTER
        DJNZ    LPA        ;CONTINUE FOR 8 BYTES
;

```

; COLOUR ROUTINE

```

;
POP      HL      ;GET ORIGINAL SCREEN POSITION
SET      S,H      ;ALIGN TO COLOUR TABLE
LD       A,(IX+03H) ;GET COLOUR INFORMATION
LD       B,08
CALL     ADDOUT
LPB:    RLCA
        RRCA      ;HANG ABOUT A BIT
        OUT      (VRM),A
        DJNZ     LPB
        RET
;

```

.COMMENT f

Routine to handle control characters within data

It assumes that:-

3 = CSR x,y

4 = PAPER n

6 = INK n

8 = Backspace

10 = Linefeed

f

CONTROL:

```

LD      HL,CONTRL
CP      3      ;IS IT CSR X,Y ?
JR      NZ,CONTA ;NO BUT CHECK IF LAST BYTE WAS
BIT     7,(HL)
JR      NZ,CONTD
BIT     4,(HL)
JP      NZ,CONF  ;MUST BE PAPER COLOUR
BIT     0,(HL)
JR      NZ,CONTA
BIT     1,(HL)
JR      NZ,CONTA

```



```

        SET    0, (HL)      ; FLAG THAT X CO-ORD EXPECTED
        SET    1, (HL)      ; AND Y CO-ORD ALSO EXPECTED
        JR     BACK

CONTA:
        BIT    0, (HL)      ; IS THIS BYTE X COORD
        JR     Z, CONTB     ; NO GO CHECK IF ITS Y COORD
        LD     (IX+00H), A   ; O.K ITS X COORD
        RES    0, (HL)      ; RESET X FLAG
        JP     BACK

CONTB:
        BIT    1, (HL)      ; IS IT A Y COORD
        JR     Z, CONTC     ; NO SO CHECK ANOTHER CONTROL
        LD     (IX+01H), A   ; SAVE Y COORD
        RES    1, (HL)      ; RESET FLAG
        JP     BACK

CONTC:
        CP     6
        JR     NZ, CONTD    ; NOT INK CONTROL BUT CHECK IF LAST WAS
        BIT    7, (HL)      ;
        JR     NZ, CONTD
        BIT    4, (HL)      ; MAKE SURE PAPER ISN'T EXPECTED
        JR     NZ, CONTD
        SET    7, (HL)      ; SET FLAG
        JP     BACK

CONTD:
        BIT    7, (HL)      ; IS FLAG SET TO EXPECT THIS TO BE
        JR     Z, CONTE     ; INK COLOUR. IF NOT CHK NXT CONTRL
        AND    A
        RLCA
        RLCA
        RLCA
        RLCA
        OR     (IX+04H)      ; PAPER BYTE MUST BE DEFINED FOR EACH PROGRAM
        LD     (IX+03H), A   ; SAVE IT
        RES    7, (HL)      ; RESET FLAG
        JP     BACK

CONTE:
        CP     10
        JR     NZ, CONTF    ; IS IT A LINE FEED
        JP     LNFED        ; GO AND DO IT BEFORE RETURNING

CONTF:
        CP     4
        JP     NZ, CONTG
        SET    4, (HL)
        JP     BACK

```


CONTG:

```

    BIT    4,(HL)
    JP     Z,BACK
    LD     (IX+04H),A
    RES    4,(HL)
    JP     BACK

```

;*****

```

;
CONTRL: DB    00H
XCORD:  DB    00H
YCORD:  DB    00H
CHAR:   DB    00H
INKCOL: DB    00H
PAPCOL: DB    01H

```

ADDOUT:

```

    PUSH    AF
    LD      A,L
    OUT     (VDP),A
    LD      A,H
    OR      40H
    OUT     (VDP),A
    POP     AF
    RET

```

ADDIN:

```

    PUSH    AF
    LD      A,L
    OUT     (VDP),A
    LD      A,H
    AND     3FH
    OUT     (VDP),A
    POP     AF
    RET

```

```

;
STKEND: DS    100H
STACK:  DW    0000H
REGSET: DB    02H,0C0H,0FH,0FFH,03H,07EH,07H,11H

```


CMP:

; Revised Compare routine V2.4

; ENTRY HL points to first operand

; DE points to second operand

; EXIT Z flag HL=DE

; NZ HL <> DE

; C flag HL < DE

; NC & NZ HL > DE

PUSH BC

LD A,3

LD C,3

; LENGTH OF ARRAYS

LD B,0

ADD HL,BC

EX DE,HL

; DE = FIRST OPERAND

ADD HL,BC

; HL = 2ND OPERAND

LD B,C

; B=LENGTH

OR A

; CLEAR CARRY FLAG

CMPLP:

DEC HL

; GET LESS SIG BYTE

DEC DE

; NOTE LESS NOT LEAST!!

LD A,(DE)

; GET A BYTE

SBC A,(HL)

JR NZ,GOB_K

; RET IF <> WITH FLAGS SET

DJNZ CMPLP

GOB_K:

POP BC

RET

DIV24:

.COMMENT /

Multi-precision divide routine (unsigned)

ENTRY HL -> dividend

DE -> divisor

EXIT HL's buffer (WRKA) holds answer

/

```

                PUSH    BC                ;PRESERVE BC

LD      B,03                ;COUNT FOR THIS PROGRAM
LD      (DEND),HL          ;DIVIDEND Address
LD      (DSOR),DE          ;DIVISOR ADDRESS
LD      C,B                ;SAVE COUNT

LD      L,C
LD      H,0
ADD     HL,HL                ;BITS = LEN*8+1
ADD     HL,HL
ADD     HL,HL                ;#8
INC     HL                  ;+1
LD      (BCNT),HL          ;LENGTH OF DIVIDEND

LD      HL,HIDE1
LD      DE,HIDE2
LD      B,C
SUB     A
DV24_B:
LD      (HL),A
LD      (DE),A
INC     HL
INC     DE
DJNZ    DV24_B

LD      HL,HIDE1
LD      (HPTR),HL
LD      HL,HIDE2
LD      (OPTR),HL
LD      HL,(DSOR)            ;HL=ADD OF DIVISOR
LD      B,C
SUB     A

DV24_C:
OR      (HL)
INC     HL
DJNZ    DV24_C
OR      A
JR      Z,ERR_R                ;EXIT IF /0

OR      A

```



```

L_24:  LD    B,C
        LD    HL,(DEND)

L_25:  RL    (HL)
        INC   HL
        DJNZ  L_25

D_E_C:
        LD    A,(BCNT)
        DEC   A
        LD    (BCNT),A
        JR    NZ,L_26
        LD    A,(BCNT+1)
        DEC   A
        LD    (BCNT+1),A
        JP    M,E_XIT          ;COUNT NEGATIVE

L_26:  LD    HL,(HPTR)
        LD    B,C

L_27:  RL    (HL)
        INC   HL
        DJNZ  L_27

        PUSH  BC
        LD    A,C
        LD    (SUBCNT),A
        LD    BC,(DPTR)
        LD    DE,(HPTR)
        LD    HL,(DSOR)
        OR    A

L_28:  LD    A,(DE)
        SBC   A,(HL)
        LD    (BC),A
        INC   HL
        INC   DE
        INC   BC
        LD    A,(SUBCNT)
        DEC   A
        LD    (SUBCNT),A
        JR    NZ,L_28

        POP   BC
        CCF
        JR    NC,L_24

```



```

        LD     HL, (HPTR)
        LD     DE, (OPTR)
        LD     (OPTR), HL
        LD     (HPTR), DE
        JP     L_24

ERR_R:  SCF                                ;FLAG INVALID RESULT
        JR     G_BK (WRKA) holds answer
E_XIT:  OR     A                            ;CLEAR CARRY RESULT OK!

G_BK:
        LD     HL, (HPTR)                  ;HL -> REMAINDER
        POP    BC
        RET

DEND:   DS     2
DSOR:   DS     2
HPTR:   DS     2
OPTR:   DS     2
SUBCNT: DS     1
HIDE1:  DS     24
HIDE2:  DS     24                        ;CAN BE UP TO 255 BYTES

MUL24:

.COMMENT /
Multi-precision Binary multiply
ENTRY HL -> points to Multiplicand
        DE -> points to multiplier

EXIT HL -> points to memory area holding answer in this case WRKA
/

        PUSH   BC                        ;PRESERVE BC

        LD     B, 03H                    ;COUNT FOR THIS PROGRAM
        LD     C, B                      ;COUNT INTO C
        LD     B, 0
        ADD    HL, BC                    ;HL POINTS TO BUFFER END
        EX     DE, HL                    ;HL=MULTIPLIER DE = BUFFER END
        LD     (MULT), HL                ;SAVE MULTIPLIER
        LD     HL, HPROD                  ;ANOTHER BUFFER
        ADD    HL, BC
        LD     (EPROD), HL              ;END OF BUFFER

;SET NUMBER OF BITS COUNT = LEN*8+1

```



```

LD      L,C
LD      H,B
ADD     HL,HL
ADD     HL,HL
ADD     HL,HL      ;*8
INC     HL          ;+1
LD      (BCNT),HL  ;SAVE NUMBER OF BITS
Z1:     LD      B,C      ;LENGTH IN BYTES
LD      HL,HPROD

Z2LP:   LD      (HL),0
INC     HL
DJNZ    Z2LP

AND     A
Z3LP:   LD      B,C
LD      HL,(EPROD)      ;GET ADDRESS LAST BYTE

Z4LP:   DEC     HL          ;ALIGN TO LAST BYTE
RR      (HL)
DJNZ    Z4LP

LD      L,E
LD      H,D
LD      B,C          ;LEN IN BYTES

Z5LP:   DEC     HL
RR      (HL)
DJNZ    Z5LP

JP      NC,DECNT
PUSH    DE          ;ADDRESS OF MULTIPLICAND
LD      DE,(MULT)
LD      HL,HPROD
LD      B,C
AND     A          ;CLEAR CARRY FLAG

ZADD:   LD      A,(DE)
ADC     A,(HL)
LD      (HL),A
INC     DE
INC     HL
DJNZ    ZADD
POP     DE          ;MULTIPLIER ADDRESS

DECNT:  LD      A,(BCNT)
DEC     A
LD      (BCNT),A
JP      NZ,Z3LP
PUSH    AF

```



```

LD      A,(BCNT+1)          ;HIGH BYTE
AND     A
JP      Z,EXITT
DEC     A
LD      (BCNT+1),A
POP     AF
JP      Z3LP

EXITT:
POP     AF
POP     BC
RET

BCNT:   DW      00
        DW      00
EPROD:  DW      00
MULT:   DW      00
HPROD:  DS      12

.COMMENT ?
16 bit multiplication routine .... ENTRY HL = Multiplicand
                                   DE = Multiplier
                                   EXIT  HL = PRODUCT

?
MUL16:  PUSH    BC          ;SAVE BC
        LD      C,L
        LD      B,H        ;BC = MULTIPLIER
        LD      HL,0        ;SET PRODUCT TO ZERO
        LD      A,15        ;BIT COUNT - 1
MULA:   SLA     E
        RL      D          ;SHIFT MULTIPLIER LEFT
        JR      NC,MULB     ;JP IF MSB OF PLIER = ZERO
        ADD     HL,BC        ;ADD MULTIPLICAND TO PRODUCT
MULB:   ADD     HL,HL        ;SHIFT PRODUCT LEFT
        DEC     A          ;DEC BIT COUNT
        JR      NZ,MULA

;NOW! IF MULTIPLIER IS 1 THEN ADD MULTIPLICAND ONCE MORE SIMPLE ?

        OR      D          ;SIGN FLAG = MSB MULTIPLIER
        JP      P,MULC      ;EXIT IF 0
        ADD     HL,BC        ;ELSE ADD TO PRODUCT
MULC:   POP     BC
        RET

```


;
;
; .COMMENT ?

16 Bit division. CALL DIV16S for signed division or MNDIV for unsigned division

ENTRY HL = Dividend

DE = Divisor

EXIT HL = Quotient

DE = Remainder

BC preserved

?

DIV16S:

```

    PUSH    BC
    LD      A,H          ;MSB DIVIDEND
    LD      (SIGN),A     ;SAVE SIGN FOR REMAINDER
    XOR     D            ;EX OR WITH MSB OF DIVISOR
    LD      (ANS),A      ;SAVE SIGN FOR ANSWER

```

DIV16A:

```

    LD      A,D
    OR      A
    JP      P,DIVA       ;DIVISOR IS POSITIVE!
    SUB     A            ;MINUS SO SUBTRACT FROM ZERO
    SUB     E
    LD      E,A
    SBC     A,A          ;INSTIGATE A BORROW (A=0FFH IF BORROW)
    SUB     D
    LD      D,A

```

DIVA:

```

    LD      A,H          ;MSB DIVIDEND
    OR      A
    JP      P,DIVB       ;IF POSITIVE
    SUB     A
    SUB     L
    LD      L,A
    SBC     A,A
    SUB     H            ;AS ABOVE
    LD      H,A

```

DIVB:

```

    CALL    MNDIV
    JR      C,ENDIV      ;IF DIVIDE BY ZERO (OH NO!)
    LD      A,(ANS)
    OR      A
    JP      P,DIVC       ;IF QUOTIENT POSITIVE JUMP!!
    SUB     A            ;SUB QUOTIENT FROM ZERO
    SUB     L
    LD      L,A
    SBC     A,A
    SUB     H
    LD      H,A

```

DIVC:

```

    LD      A,(SIGN)     ;CHECK SIGN OF REMAINDER
    OR      A
    JP      P,ENDIV      ;AND NEGATE IF NEGATIVE
    SUB     A            ;IF REMAINDER POS JUMP TO RETURN
    SUB     E

```



```

LD      E,A
SBC     A,A
SUB     D
LD      D,A
JR      ENDIV

MNDIV:
;UNSIGNED DIVISION CALLED FROM HERE
PUSH    BC
LD      A,E          ;CHECK FOR DIVIDE BY ZERO
OR      D
JR      Z,BFAL
LD      A,L
OR      H
JR      NZ,DOIT
BFAL:   LD      HL,0
LD      D,H
LD      E,L
SCF                     ;SET CARRY TO DENOTE /0
POP      BC
RET

DOIT:
LD      C,L
LD      A,H
LD      HL,0          ;REMAINDER TO ZERO
LD      B,16          ;BIT COUNT
OR      A              ;CLEAR CARRY FLAG

MDIVLP:
;32 BIT SHIFT LEFT    CARRY ->C->A->L->H

RL      C
RLA
RL      L
RL      H              ;CLEARS CARRY COS HL=0
PUSH    HL
SBC     HL,DE
CCF                     ;COMPLEMENT SO THAT 1= SUCCESSFUL SUBTRACT
JR      C,CLRSTK      ;JUMP IF REMAINDER >= DIVIDEND
EX      (SP),HL        ;OTHERWISE RESTORE REMAINDER

CLRSTK:
INC     SP
INC     SP              ;CLEAR STACK
DJNZ    MDIVLP
EX      DE,HL          ;SHIFT LAST CARRY IN QUOTIENT DE = REMAINDER
RL      C
LD      L,C            ;LSB  ANSWER
RLA
LD      H,A            ;MSB  ANSWER
OR      A              ;CLEAR CARRY TO SHOW VALID DIVIDE
POP     BC
RET

ENDIV:  POP     BC
RET                     ;TO CALLER

ANS:    DB      00H
SIGN:   DB      00H

```



```

;*****;
ASCBIN:
;*****;
;ASCII TO SP BINARY
;Converts an Ascii number held in the buffer ASCBF to a SP Binary number
;which is left in WRKA LSB  WRKA+1 NSB  WRKA+2 MSB  All registers corrupted

```

```

;*****;

```

```

                LD    HL,WRKA+3    ;WORK AREA
                XOR    A
                LD    B,4
ZBF:            LD    (HL),A
                DEC    HL
                DJNZ   ZBF
                LD    HL,ASCBF    ;ASCII BUFFER TERMINATED
                LD    A,(HL)      ;WITH A
                CP     2FH
                JP     C,ERIN
                CP     40H
                JR     NC,ERIN
                CP     0FFH      ;<CR>
                JP     Z,ACFIN
                SUB    30H      ;NOW BINARY
                PUSH   HL      ;SAVE ASCBF
                PUSH   AF      ;SAVE 1ST DIGIT
                LD     HL,WRKA   ;ZEROED ABOVE
                CALL   SPLD     ;PUT IT INTO C/DE
ACLP:           SLA     E
                RL      D
                RL      C
                LD     HL,WRKA   ;NOW #2 = YTHAD1
                CALL   DWNLD     ;PUT IN WRKA
                SLA     E
                RL      D
                RL      C      ;#4
                SLA     E
                RL      D
                RL      C      ;#4
                LD     HL,WRKA
                CALL   REGAD     ;ADD WRKA = #10
                POP     AF      ;GET DIGIT
                CALL   REGAD1    ;ADD IT TO VALUE
                POP     HL      ;GET ASCBUF BACK
                INC     HL
                LD     A,(HL)
                CP     0FFH
                JR     Z,ACFIN
                SUB    30H      ;NOW BINARY
                PUSH   HL      ;SAVE ASCBF
                PUSH   AF      ;SAVE DIGIT

```



```

                JR      ACLP
ACFIN:          LD      HL,WRKA
                CALL    DWNLD      ;BINARY INTO WRKA
                RET
ERIN:           XOR A
                RET

BINASC:

;*****
;S.P BINARY TO ASCII
;*****
;Converts a SP binary number to Ascii terminated with a single 0FFH byte
;which is left in ASCBF   Number must be posted to WRKA before entry....
;SP NO IN BC/DE
;DE = DIVISION TABLE
;ASCII STORED IN ASCBF TERMINATED FF

                LD      DE,DIVTAB      ;TABLE OF CONSTANTS
                LD      HL,ASCBF
                XOR     A
CONLP:          CCF                    ;CARRY = 1ST TIME SWITCH
                PUSH    AF
                PUSH    HL
                PUSH    DE
                LD      HL,WRKA        ;PUT
                CALL    SPLD           ;PUT SP INTO BC/DE
                POP     HL             ;HL = DIVTAB
                LD      B,2FH          ;ASCII (30H - 1) = "0" - 1
INKASC:         INC     B              ;INCREMENT ASCII CHARACTER
                LD      A,E            ;LSB
                SUB     (HL)
                LD      E,A
                INC     HL
                LD      A,D            ;NSB
                SBC     A,(HL)
                LD      D,A
                INC     HL
                LD      A,C

```



```

SBC    A, (HL)
LD     C, A
DEC    HL          ;REALIGN TO 100,000 ETC
DEC    HL
JR     NC, INKASC   ;LOOP UNTIL 100,000
CALL   REGAD        ;ADD 100,000 TO MAKE POSITIVE
INC    HL          ;BUMP TO NEXT -> 10,000
CALL   PUTBK        ;PUT REMAINDER BACK FROM CURRENT VALUE
EX     DE, HL       ;DE = DIVTAB(10,000) ADDRESS
POP    HL          ;HL = PBUF = ASCBF
LD     (HL), B      ;SAVE ASCII DIGIT
INC    HL          ;NEXT PRINT AREA
POP    AF          ;GET CARRY SWITCH
JR     NC, INKDIV    ;SEC PASS NO OVERFLOW IF ANY HANDLED
DEC    HL
LD     (HL), "0"    ;SET MSB OF ANSWER
JR     OVE2
OVE1:  INC    (HL)
LD     A, B
SUB    10

LD     B, A
OVE2:  LD     A, "9"
CP     B
JR     C, OVE1      ;MORE OVERFLOW!!
INC    HL
LD     (HL), B      ;NO. OF HUNDERD THOUS
INC    HL
CCF
JR     CONLP
INKDIV: INC    DE    ;WHEN ROUTINE FALLS THROUGH
INC    DE    ;WE HAVE DIVIDED BY 1000,000, 10,000
;SO BUMP TO 10,000 WHICH IS
;INTERGER DIVISION
LD     A, 04        ;NO OF DIGITS
JR     NXTASC

```

;THIS ENTRY TO BE USED FOR INTEGER TO ASCII
;MUST LOAD HL WITH ACSBF BEFORE ENTRY

```

INTEG:
LD     DE, DIVT2    ;TABLE OF INTEGER CONSTANTS
LD     A, 5         ;NO OF ASCII DIGITS
NXTASC:
PUSH   AF          ;SAVE NO OF DIGITS
PUSH   HL          ;SAVE PRINT BUFFER
EX     DE, HL       ;HL = DIVT2
LD     C, (HL)      ;PWR 10 IN BC
INC    HL
LD     B, (HL)      ;MSB IN PWR
PUSH   BC          ;SAVE IT

```



```

INC     HL           ;NXT VALUE IN PWR TABLE
EX      (SP),HL      ;HL = VALUE IN BC
EX      DE,HL        ;DE = VALUE
LD      HL,(WRKA)    ;HL = CURRENT VALUE (INTEGER)
LD      B,2FH
INKAS2:
INC     B           ;DIVIDE CURRENT VALUE
LD      A,L         ;BY POWERS OF 10 STARTING
SUB     E           ;AT 10,000. REMAINDER FROM
LD      L,A         ;EACH DIVISION AND ADDED
LD      A,H         ;TO THE DIVISION AND SUM
SBC     A,D         ;= DIVIDEND FOR NEXT DIVISION
LD      H,A         ;COMPOUND SUBTRACTION
JR      NC,INKAS2   ;IN OTHER WORDS!
ADD     HL,DE
LD      (WRKA),HL
POP     DE           ;NXT PWR 10
POP     HL           ;HL = ASCBF
LD      (HL),B
INC     HL
POP     AF
DEC     A
JR      NZ,NXTASC
LD      A,0FFH

LD      (HL),A      ;TERMINATOR
RET     ;RETURN TO CALLER

DIVTAB:
DB      0A0H,06H,01,10H,27H,00
DIVT2:
DB      10H,27H,0E8H,03,64H,00
DB      0AH,00,01,00
WRKA:
DS      4
PUTBK:
EX      DE,HL       ;HL = DE : DE = DIVTAB
LD      (WRKA),HL   ;LSB
LD      HL,0
LD      L,C
LD      (WRKA+2),HL ;MSB
EX      DE,HL
RET

```


REGAD:

```

;ENTER TO POINTS TO MEMORY AREA
;ADDS C/DE WITH (HL) >
;EXIT HL >ND BUF
;C/DE HOLD VALUE

```

```

LD     A, (HL)
ADD    A,E
LD     E,A
INC    HL
LD     A, (HL)
ADC    A,D
LD     D,A
INC    HL
LD     A, (HL)
ADC    A,C
LD     C,A
RET

```

```

;ADDS VALUE IN A REG TO C/DE

```

REGAD1:

```

ADD    A,E
LD     E,A
LD     A,0
ADC    A,D
LD     D,A
LD     A,B
ADC    A,C
LD     C,A
RET

```

SPLD:

```

;ENTER HL POINTS TO START OF BUFFER WHERE VALUES ARE
;EXIT: BUFFER UNTOUCHED
;C/DE = VALUE

```

```

LD     E, (HL)
INC    HL
LD     D, (HL)
INC    HL
LD     C, (HL)
INC    HL
RET

```


DWNLD:

```
;HL = MEMORY AREA
;BC/DE = VALUES
;EXIT C/DE UNTOUCHED
```

```
;MEM = VALUE
```

```
LD      (HL),E
INC     HL
LD      (HL),D
INC     HL
LD      (HL),C
RET
```

REGSUB:

.COMMENT ?

ENTRY: HL points to memory area.

Routine subtracts REGISTERS from MEMORY HEM-REG

EXIT: C/DE holds answer.?

```
LD      A,(HL)
SUB     E
LD      E,A
INC     HL
LD      A,(HL)
SBC     A,D
LD      D,A
INC     HL
LD      A,(HL)
SBC     A,C
LD      C,A
RET
```

WRKAFL:

;Fill area of memory pointed to by HL with whats in area pointed to

;by DE

;ENTRY HL = area to be filled

; DE = area containing data

```
LD      A,(DE)
LD      (HL),A
INC     HL
INC     DE
LD      A,(DE)
LD      (HL),A
INC     HL
INC     DE
LD      A,(DE)
LD      (HL),A
RET
```


PRNUM:

;New routine to take binary out of variable and convert into Ascii &
 ;then print to display all in one go!

;ENTRY HL points to Variable

```

PUSH    BC
PUSH    DE
PUSH    HL

CALL    SPLD
LD      HL,WRKA
CALL    DWLND
CALL    BINASC
CALL    NMPRNT

POP     HL
POP     DE
POP     BC
RET

```

ALIGN:

;ENTRY C CONTAINS EXPECTED NUMBER OF CHARACTERS ..

```

LD      HL,ASCBF
LD      C,6
LD      B,0           ;COUNTER
ALLOP:  LD      A,(HL)
        CP      0FFH
        JR      Z,ALFIN
        INC     B
        INC     HL
        JR      ALLOP
ALFIN:  LD      A,C
        CP      B           ;SUB CHARACTERS
        RET     Z           ;FROM EXPECTED COUNT
        SUB     B           ;IF <> THEN INC CURSOR
ALLOPX: INC     (IX+00H)
        DEC     A
        CP      0
        JR      NZ,ALLOPX
        RET

```


Pushall Registers

;Push all registers at once. The idea is to use this routine as a subroutine
 ;But ! How do we find our return address if we do ?
 ;The trick is in the EX(sp),Hl

```

;:::::::::: program flow
;call svereg push registers
;:::::::::: program flow
;call getreg retrieve registers

```

```

S/VEREG:
    EX (SP),HL      ;HL now on stack. Return address in HL.....
    PUSH AF
    PUSH BC
    PUSH DE
    PUSH IX
    PUSH IY
    PUSH HL        ;Return address now on top of stack..
    RET
GETREG:
    POP HL          ;Get Return address <.....
    POP IY
    POP IX
    POP DE
    POP BC
    POP AF
    EX (SP),HL      ;Original value now in HL. Ret adr on stack<...
    RET

```


COMPARISONS

IF A = B	==>	JR Z
IF A <> B	==>	JR NZ
IF A => B	==>	JR NC
IF A < B	==>	JR C
OR	==>	JP M

TESTING REGISTERS

```

inc a
dec a
jp m,negative ;Register contains negative value

```

```

inc b
dec b
jp p,positive ;Register is positive

```

Test for #FF

```

inc reg
jr z,#FF

```

Test for 1

```

dec reg
jr z,=1

```

Test for 0

```

inc reg
dec reg
jr z,=0

```

Test for non-zero

```

inc reg
dec reg
jr nz,not zero

```


Clear Carry Flag

```

        or    a
or ...   and    a

```

Clear Accumulator

```

        xor    a
or ....   sub    a

```

String Comparisons

```

;Entry DE = string A
;      HL = string B
;Exit  Z flag set if equal
;Strings must be terminated with #FF

```

strglp:

```

ld    a,(de)    ;String a
cp    #ff       ;End of string terminator
ret    z
cp    (hl)      ;String b
inc    hl
inc    de        ;bump to next character
jr    z,strglp  ;test next char if previous was =
ret

```

Bubble Sort

Bsort:

```

ld    ix,table  ;start of sort tables
ld    b,n       ;where n = no of items - 1

```

bloop:

```

ld    a,(ix)    ;low
cp    (ix+1)    ;high
call  nc,switch ;is low>high ?
inc    ix       ;bump to next item
djnz  bloop
or    a         ;zero ? if not
jr    nz,bsort  ;then do it ALL AGAIN

```

switch:

```

ret    z        ;no need to switch they're =
ld    c,(ix+1)  ;now switch them
ld    (ix),c    ;both around
ld    (ix+1),a
xor    a        ;set still sorting flag
ret

```