ADVANCED PROGRAMMING TECHNIQUES ON THE AMSTRAD CPC 464

KEITH HOOK

ADVANCED PROGRAMMING TECHNIQUES ON THE AMSTRAD CPC 464

ADVANCED PROGRAMMING TECHNIQUES ON THE AMSTRAD CPC 464

KEITH HOOK

PHOENIX PUBLISHING ASSOCIATES LTD
BUSHEY, HERTFORDSHIRE.

First published in Great Britain in 1985 by
PHOENIX PUBLISHING ASSOCIATES LTD.
14, VERNON ROAD, BUSHEY, HERTS. WD2 2JL

ISBN 0 9465 7632 7

Printed in Great Britain by
Garden City Press, Letchworth
Cover Design by
Ivor Claydon Graphics
Typesetting by First Page Ltd, Watford.
Production by Denis Gibney Graphics, Chesham

CONTENTS

To Patricia, my wife,
for her undying support.

INTRODUCTION

The Amstrad CPC 464 has been designed to allow the programmer access to an excellent and powerful programming language: **Locomotive Basic**.
This book is written in the hope that it will be of use to programmers who want to get the most from their computers . In essence this book is meant to be a cookbook of ideas developed around Amstrad's Basic interpreter.

This book is not designed to teach the novice all aspects of Basic programming but supplement an assumed level of knowledge gleaned from the instruction manual.

If you have at least a nodding aquaintance with Basic, this book will allow you to progress through the numerous examples, to an advanced level of programming by showing you how to interface simple machine code routines to your Basic programs. Don't worry if you don't understand all the instructions that are described - **jump in at the deep end!** The best way to learn is by practical example; the understanding will follow later.

Within the pages of this book you will find a considerable amount of reference material that you may choose to study now, or at a later date, and you won't need a degree in computer studies to apply your new found knowledge.

The material in this book will show you how to trap the powerful routines lying deep within the Amstrad's ROM. The majority of these routines can be harnessed from

Basic, and will allow you to produce programs that are a mixture of machine code and Basic. This hybrid way of programming the Amstrad will enable you to do things from basic you thought were impossible - **Load a machine code program into an Array and run the array, or give the CPC 464 8 sprites to control**. This book will show you how.

There is nothing difficult or mysterious about the methods suggested. Of course, there are rules, and some of you will not yet have attempted to take the giant step beyond Basic but with a little application all things are possible. I reiterate, jump in with both feet !

Although it is not the intention of this book to teach Assembly language, it will give you a good insight, and I hope, a push toward investigating a completely new world of programming.

In order to take advantage of the computer's operating system you must have a good knowledge of how the computer operates, and therefore some of you may think the opening chapters a little elementary. However, I urge you to read them, you never know, you may even discover you don't know it all !

Chapter One

Basic BASIC

I am not going to insult your intelligence by starting this book with long explanations on how to wire up the plug, or connect the monitor - if you've bought the computer there is no doubt you are already using it. However, to really get to grips with what we are about, you need to understand **Basic**, it's commands, and how the **Operating System** works.

Basic programs are composed of **Basic Lines**, and **Basic Lines** are made up from **Basic Statements.** The CPC464 allows you to program with **Multi-statement Basic Lines.** A **statement** is a command that tells the computer to carry out some specific action. The Amstrad contains, within it's **ROM [Read Only Memory],** a **Basic Interpreter** which translates [interprets] each Basic statement into **Machine Code instructions.** The **Z.80** processor used by the CPC464 can perform these machine code instructions in millionths of a second which means that Basic statements are interpreted very rapidly.

Basic can never match the execution speeds of **machine code programs.** On the other hand, writing and debugging machine code is not easy. Even the smallest program can take hours to de-bug, and the slightest error can send your program on a journey to nowhere and leave you staring at a blank screen. To efficiently de-bug **machine language routines** it is sometimes necessary to play Computer with your code, testing each section with pen and paper until the area of the mistake is located.

With most programs a happy medium can be struck. The bulk of the program can be written in **Basic** with **machine code subroutines** taking care of the program sections that need extra speed. For instance, the speed of a **sort routine** can be increased by a factor of **1000** by writing the actual sort in machine code. The latter is only one example: graphics,animation, and sound can also be greatly enhanced by this method of programming. You can also perform many more functions by writing sections of **machine code** to interface with your **Basic program** - the Amstrad doesn't support sprites but by the time you finish this book you will be using them on your computer!

Overview of Locomotive Basic

You can input into your computer in two ways: **Program Mode** and **Direct Mode**.

In the **Direct Mode** you can enter commands direct from the keyboard, and as soon as you press the **ENTER** key the command will be executed immediately.

```
CLS:X=3:Y=X+1:PRINT Y <ENTER>
```

If you have typed the above line correctly the screen should now be displaying the answer **4.** Examples of other direct commands are:

RUN,NEW,AUTO.

The **Programming Mode** differs in that commands are entered into the computer prefixed by a **Line Number**. Pressing the **ENTER** key inserts the line into the current program without executing the instructions.

```
AUTO        10,10        <ENTER>          [Direct   Mode]
```

```
10 CLS

20 X=3:Y=X+1

30 PRINT Y
```

If you now **RUN** the program you will see that the end result is exactly the same as in the **Direct Mode** with the screen displaying the answer.

When you typed the **Direct Command, RUN**, Basic automatically stepped through the program lines performing each task as it was encountered. Basic always executes a program line number by line number in the correct order except when a **GOTO or GOSUB** causes it to branch to another higher or lower line number. The program will continue to run until it either runs out of line numbers, or an **END** statement is encountered which will cause the program to terminate.

```
10 CLS

20 INPUT " NAME PLEASE ";N$

30 PRINT "HELLO ";N$

40 INPUT "DO YOU LIKE YOUR CPC464 ";AN$

50 IF AN$ = "YES" THEN GOTO 80

60 PRINT "OH DEAR ! I'M SORRY ABOUT THAT ";N$

70 GOSUB 120: GOTO 100

80 PRINT "I'M VERY PLEASED YOU DO ! "

90 PRINT "BYE ";N$ : GOSUB 120
```

```
100 CLS

110 END

119 REM DELAY LOOP TO STOP SCREEN CLEARING TOO FAST

120 FOR I = 1 TO 500

130 NEXT

140 RETURN
```

This short program illustrates **program flow** within a computer. **PROGRAMS FLOW FROM START TO FINISH WITH STATEMENT LINES NUMBERED IN ASCENDING ORDER.** It is possible to alter the flow of the program as we have done in line 50 by testing if the answer **[AN$]** to the question was **YES.** If the answer was positive program flow was re-directed to line 80 and skipped over lines 60 & 70.

In the previous program we have used lines numbered in multiples of 10. Why 10 ? Well, we could have used any line numbering, **1,2,3,4 or 1,3,5,7**, it is, however, advisable to keep line numbering simple and uncluttered. If you use multiples of 10 you can always add further lines if you find it necessary. After adding lines to your program the line numbers will not be in even increments but on the Amstrad you can always 'tidy' the program by using the **RENUM** function. Go on, try it. Type **RENUM 100,10,5** ‹**ENTER**›. Have you noticed that when you now list the program the line numbers have changed to 100,105,110.... and are incremented in multiples of 5, and the program starts with line number 100 ?

You will use within your programs two types of operands: **CONSTANTS** and **VARIABLES.**

Constants are set values that never change. The value of **Pi** is a **constant.** There are two types of **constants: Integer** or non-decimal figures like 1,12,100,1111, and **Real constants** like 3.333, 1.00789.

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 numbers and character strings. The Amstrad allows you to specify a variable name up to a length of 40 characters. Examples of variable names are **X,Y,A5,BA,CAGE,NAME.** Locomotive Basic also allows the programmer to specify the type of variable at the start of the program: **INTEGER [%],REAL [!],** and **STRING [$].**

Integer Variables hold numbers in the range **-32768 to 32767** and can be created by **DEFINT** statement, or suffixed with % as in **X%.**

Real Variables are created using **DEFREAL** or created within the program by **!.** Examples of **Real Variables** are: $1.35E^{+38}$ [$1.35*10^{+38}$], $6.35E^{-20}$ [$6.35*10^{-20}$]. The number range of this type of variable is large enough to suffice even the most hardened number cruncher. [$1.7E^{+38}$ down to $2.9E^{-39}$].

Real Numbers use up **five bytes** of memory space. This observation is important. The **default assignment for all variables is REAL.** You should get into the habit of **DEFINING** your variables at the very start of your program - unless you are going to use variables that require a high degree of accuracy then use **DEFINT.** E.g.

```
DEFINT A,C,E-N,P-Z
```

The above example will result in **all** variables with the exception of **B,D,O** being **DEF**ined as **Integer.** The computer can operate many times faster on **integers**, and so your programs will **RUN FASTER!**

Another type of variable is the **String Variable**. In a nutshell they are simply strings of alphabetical ,numerical, or special characters that have some significance within the computer. **Strings** are donated by placing the Dollar sign **[$]** after the variable name: **E$="ME":L$="YOU":X$=CHR$(13)..**

The CPC464 is capable of manipulating strings in many ways and the ability to handle strings provides one of the more powerful functions of the computer.

Another type of variable used within 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. As you become more proficient in your programming you will find yourself using this type of variable more and more.

A typical example of a **subscripted variable** is: **X(1)** where **X is the variable with a subscript of one.** Another example is: **X(6)** meaning **X subscript 6**. All subscripted variables that carry the same name e.g. **A(1),A(2) .....A(20) etc** constitute an **Array.**

Arrays are created by **DIM**ensioning them at the start of your program - this is not strictly true, but it is good programming practice to dimension arrays at the start of your program - e.g **DIM X(12)** would **dimension** an **array** from **X(0)** through to **X(12).** The subject of arrays is too important to dismiss in a few short lines, and we shall discuss it fully in a later Chapter.

Locomotive basic includes a complete set of built in **Functions**. These **functions** can perform certain specialised computations: **RANDOM** and **RND** allow generation of random numbers - essential for games and simulation. **Mathematical functions** include: **FIX (truncation), CINT(integer), INT(whole number), SIN(Sine of angle) ,CREAL(converts to real number).**

Error checking in programs is taken care of by **ERROR,ERR,ERL, ON ERROR GOTO**, and **RESUME.** Used in concert these commands let you test your error trapping routines by simulating an error condition with **ERROR** while **ON ERROR GOTO** sets up the line number of your error routine, and **RESUME**, resumes program execution after an error has occurred.

The computer includes a **Line Editor** which is invoked with the **EDIT** command. The **Editor** allows each individual line to be called up, the cursor positioned within the line, and characters can then be removed, corrected or inserted. A **COPY** facility further enhances the **Edit Mode** by allowing the characters under the **Copy Cursor** to be copied, and inserted at the **Edit Cursor** position. **TRON** and **TROFF** (Trace On,Trace Off), turn the **TRACE** facility on, or off. **TRON** displays each line number on the screen as it is executed, thus allowing you to examine program flow. You can test this for yourself by giving the **direct command ›TRON ‹ENTER›.** Now **RUN** program **number 2.** Answer the prompts with "Yes" then re-**RUN** the program, but this time answer the prompt with "No".

Did you notice how the program flow was redirected when you ran the program a second time ? **TRON** is a very valuable debugging aid when you have a large program in memory, and it doesn't respond as you expected it to.

STOP allows you to insert a **Break Point**, and stop program execution at any point. After **STOP** you can examine the contents of variables,arrays etc., while **CONT** will continue program execution after the **STOP**, providing the program has not been altered.

CLEAR sets **all** variables to zero, and all strings to " " [null]. Arrays are also erased, and all loops are abandoned.

NEW deals a death blow to any resident Basic program, and wipes it from memory then re-initialises the Basic Interpreter.

Chapter Two

REPRESENTING MEMORY LOCATIONS

In this chapter we are going to consider the subject of **decimal, binary,** and **hexidecimal** numbers, and how they are stored in the Amstrad computer. No! Don't skip this section - the subject is far easier to understand than you might think. A little time spent learning the ground rules will be rewarded a hundred fold in your later attempts at programming.

BINARY NUMBERS

A memory location in the Amstrad computer has an "address" - the address can be any integer from 0 through to 65535. Memory locations store data and commands that are 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 with **binary numbers** - all your decimal and hexidecimal 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 which represent a state of **ON [1]** or **OFF [0]**. The computer recognises the **ON\OFF** patterns and acts accordingly.

If you understand how decimal,hexidecimal, and binary numbers interact you will be capable of writing more efficient programs, and performing operations from Basic that are not mentioned in the manual. E.g. Bit testing, or compacting data, to name but two.

One memory location can store 8-binary digits. One binary digit is called a **bit** , and **8 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 x 100 + 5 x 10 + 2 x 1**, and we can also write 152 in it's **expanded form** by expressing it as, $\mathbf{1 \times 10^2 + 5 \times 10^1 + 1 \times 10^0}$, 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 :-

```
[2 x 2] + [2 x 1] + 1   or   1 x 2² + 1 x 2¹ + 1 x 2⁰  = 7
```

In binary, each position represents a **power of 2** rather than a power of 10.

```
1011 = 1 x 2³ + 0 x 2² + 1 x 2¹ + 1 x 2⁰
     =   8    +   0    +   2    +   1    = 11₁₀
```

The 8 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_{10} =

$$1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1$$

= [2 x 2] 7 times + [2 x 2] 6 times + 2 x 2 + 2 + 1

= 128 + 64 + 4 + 2 + 1 = 199

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	8	4	2	1
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.**

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	9	8	7	6	5	4	3	2	1	0
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	1
254	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0
256	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 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 DEFINT I      ' I IS NOW AN INTEGER VARIABLE
30 INPUT " INPUT ANY NUMBER "; I
40 PRINT I
50 GOTO 30
```

Run the program and try answering the prompt with different values for I. Now try giving I a value greater than 32767. What happened? The computer responded with **O/V ERROR** meaning that you tried to assign a number to integer I that was out of range.

Table 2.2

POWERS OF 2

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 **reset** to the **off** position. This terminology is still used today - if a bit is **1** we say the bit is **set**, and if a bit is **0** we say the bit is **reset.**

We shall now be dealing with three number systems: **Binary,Decimal,and Hexidecimal.** To distinguish between the three systems we shall use the following subcripts: $binary_B$, $decimal_D$, $hexidecimal_H$. From now on, if you see $\mathbf{32456_D}$ you will know the number is **decimal,**and $\mathbf{AC00_H}$ is **hexidecimal.**

BINARY ARITHMETIC

Binary addition and **multiplication** are very easy operations to understand. As there are only two digits (0 & 1) to deal with in binary, you only have **four permutations** to learn for addition, and four for multiplication.

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

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

From the above you can see that the only rule to remember is **1+1 = 0 and carry 1.**

Addition

Carry line 1 1 1 0

15_D = 1 1 1 1

6_D = 0 1 1 0

Answer ..... 21_D = 1 0 1 0 1

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

Binary addition table

+	0	1
0	0	1
1	1	10

Multiplication

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

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

Binary multiplication Table

X	0	1
0	0	0
1	0	1

```
Multiplicand ..............  1 0 1 0      =    10D

Multiplier ................  0 1 1 0      =     6D
                             -------

                             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        =     60D
                      ============
```

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

Fact 2: On each step the partial answer is shifted one place to the left - even if the multiplier contains an 0.

Fact 2 provides us with a quick method of multiplying binary numbers for powers of 2 (2,4,8,16 etc).

```
  2 x 2 = 4     2 = 0000 0010  shifted one place left = 0000 0100 = 4

2 x 8 = 2 x 2 x 2   2 = 0000 0010 shifted two places left = 0000 1000 = 8
```

Binary division

Binary division is simplicity itself as you can see at a glance if one number will divide into another. Division can also be performed by successive subtraction until a negative remainder is encountered. **Division is the inverse of multiplication**, so an easy way of dividing with multiples of 2 is to use the inverse of **Fact 2**, i.e. **shift one place right.**

```
8 / 4 = 2 .. 4 = 2 x 2   so shift 2 places right

      8D = 0000 1000

      shifted right once  =   0000 0100  = 4D

      shifted right twice =   0000 0010  = 2D
```

LOGICAL OPERATIONS

An often neglected feature of the computer is its ability to perform **logical operations**. Amstrad's Basic instruction set is very powerful and provides a full set of logical operations that can be called on from Basic.

In **Boolean Algebra** there can be only two answers, or states: **TRUE [1]** and **FALSE [0].** If the result of a logical operation is true the computer sets the byte to all ones. If the result is false the 8 bits are reset to zeros.

Logical operations are not hard to grasp, after all, we use the **AND** statement frequently within our Basic programs.

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

Whenever you program the computer with an **IF, THEN, ELSE** statement you are actually using a logical expression.

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 Amstrad Basic. Whenever **G<>0** the

program will always jump to line 100. **IF G AND 16** is also allowed - this very useful shorthand will save a lot of typing.

The use of **AND,OR,** and **NOT** is not restricted to simple relational expressions as mentioned above. **AND,OR**,and **NOT** can also be used for **Boolean** operations, bit manipulation, and bit comparision.

NOT

The **NOT** expression forms the complement of the number by inverting each bit of 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 bits of a byte.

```
12 AND 4  =>  = 4
```

```
          0000 1100  = 12D
AND       0000 0100  =  4D
          ---------------
          0000 0100  =  4D
```

```
25 AND 12  =>  =  8D
```

```
          0001 1001  = 25D
AND       0000 1100  = 12D
          ---------
          0000 1000  =  8D
```

```
4 AND 2  =>  =  0D
```

```
          0000 0100  =  4D
AND       0000 0010  =  2D
          ---------
          0000 0000  =  0D
```

You can test the above result for yourself using Basic.

```
PRINT 4 AND  2   <ENTER>
```

OR

The **OR** operation is frequently used to set certain bits without affecting other bits of the byte(s).

```
4 OR 2  =>  = 6
```

```
          0000 0100  = 4D
OR        0000 0010  = 2D
          ---------
          0000 0110  = 6D
```

```
-1 OR -2  =>  = -1
```

```
          1111 1111  = -1D
OR        1111 1110  = -2D
          ---------
          1111 1111  = -1D
```

XOR (Exclusive OR)

XOR is often used to set a byte to zero.

```
3  OR  3  =>  =  0
4  OR 2  =>  =   6
```

```
          0000 0100  = 4D
XOR       0000 0010  = 2D
          ---------
          0000 0110  = 6D
```

Table 2.3
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

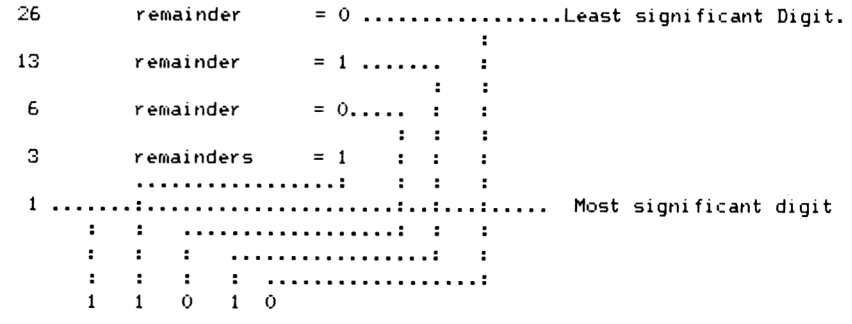
Table 2.4
LOGICAL OPERATIONS

```
        NOT TRUE  = FALSE

        NOT FALSE = TRUE
TRUE  AND TRUE  = TRUE

TRUE  AND FALSE = FALSE

FALSE AND TRUE  = FALSE

FALSE ANS 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 OR FALSE  = FALSE
```

DECIMAL TO BINARY

It is sometimes necessary to convert from decimal to binary. The process is simply a matter of repeatedly dividing by 2 and storing the remainder in a special column.

Convert 26 to binary



The division is continued until we get a 0 or a 1 as the final answer. The binary number is obtained by placing the final result in the left most column follwed by the remainders read from bottom to top. Easy isn't it ?

If you are unfamiliar with the binary system **Listing One** will help you to see how binary numbers relate to decimal values.

LISTING ONE:

```
1  MODE 1:PAPER 0:CLS  ' BOOLEAN BIT SETTING

10 WINDOW #1,8,30,3,3

20 WINDOW #2,8,34,5,5

30 WINDOW #3,8,34,7,7
```

```
 40 WINDOW #4,8,34,8,8

 50 LOCATE #1,1,1

 60 CLS#1:INPUT#1,"INPUT YOUR NUMBER";A

 70 LOCATE #2,1,1

 80 PEN#2,2:INK 2,5:PRINT#2,"YOUR NUMBER = ";A

 90 LOCATE #3,1,1:PEN#3,5:INK 5,24

100 PRINT #3,"BIT   0  1  2  3  4  5  6  7 "

110 LOCATE #4,6,1:PEN#4,6

120 FOR I= 0 TO 7

130 IF A AND 2^I THEN PRINT#4,"1 "; ELSE PRINT#4,"0 ";

140 NEXT

150 GOTO 50
```

HEXIDECIMAL NUMBERS

Once you have learned the rudiments of the binary system, hexidecimal numbers are not difficult. In hexidecimal we need the digits **0 to 15**. Normal conventions substitute **A B C D E F** for the digits 10 to 15. In this notation, the decimal number **13** becomes $\mathbf{D_H}$.

Hexidecimal numbers allow us to manipulate binary numbers in a more manageable fashion. Large binary numbers look 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 default address of the Amstrad's screen. In decimal this address is 49152_D, and in binary, 1100 0000 0000 0000. In hexidecimal the address is $\mathbf{C000_H}$.

It is obvious from this example that the easiest number to remember is the hexidecimal form.

Two **HEX** (short for hexidecimal) **digits make up one byte (8 bits),** and **one hex digit makes up 4 bits** (often called a nibble). By splitting our binary numbers into groups of 4 it is relatively simple to convert them to hexidecimal notation.

Table 2.5

HEXIDECIMAL 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_D** If you look at the hex values in **table 2.5** you can see how to convert the value to hex by equating each set of 4 bits to the equivalent hex digit.

```
          E     C     4     D

       1110  1100  0100  1101

= 16³ x 14  +  16² x 12  +  16¹ x 4  + 13
= 60493D
```

Hexidecimal addition

Addition with hexidecimal numbers is very similar to adding decimal numbers, except that we are counting to 16 before carrying 1 into the next column left.

```
100    .....Carry line.....  1110
AC01H                        DACBH
1FC3H                        2BB0H
------                       -----
CBC4H ......Answer.........  1067BH
```

To help you with your hexidecimal addition, take a look at **Table 2.6**. If you want to add C_H to B_H 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, and you will find the answer is 17_H. If you were actually adding these two digits, you would put 7 in the answer line and 1 in the carry line.

Table 2.6
HEXIDECIMAL 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

Let us now consider how the Amstrad actually stores the numbers in it's memory.

All Z.80 based computers store their numbers in what is termed **Least Significant Byte [LSB], Most Significant Byte [MSB]** format. 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 a memory saving of more than 60%.

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 **Msb** into 65451. To find the **Lsb & Msb** values we can use the following, easy to learn, formula:

```
Msb = INT (Number/256)

Lsb = Number - 256 * Msb
```

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

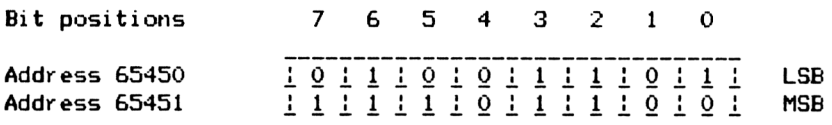
```
Msb = INT(60493/256)      = 236D  = ECH

Lsb = 60493 - 256 * Msb = 77D = 4DH
```

If we were working in Basic, we could put this number into memory by using the **POKE** command.

```
POKE 65450,Lsb

POKE 65451,Msb
```

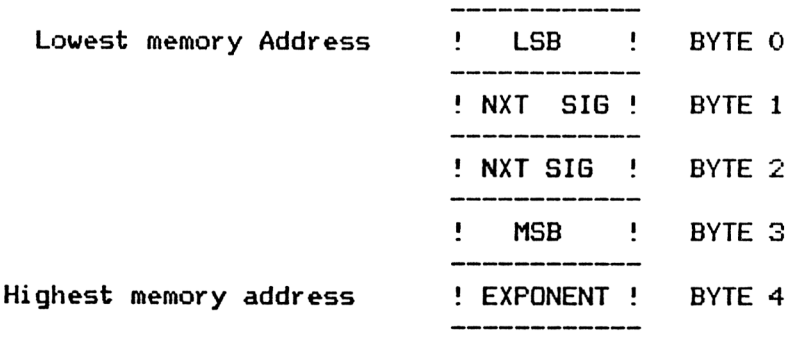
It is important to understand this rule. Later, when we start **Poking** machine code instructions into memory, we must know how to store and retrieve them correctly.

How a two byte number is stored in memory:



Earlier, we said that **REAL variables** used **five bytes** of memory; this type of variable is stored in memory in exactly the same manner as integers, but obviously, the number is spread over five bytes.

Memory allocation for REAL numbers



The **exponent** portion is expressed as a power of **2,** allowing the range 10^{+38} to 10^{-39}.

NEGATIVE VALUES

Up to this point we have been discussing 8 and 16-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 could not perform normal mathematical functions. Clearly, to work with positive and negative values, we need some way to tell which integer is positive, and which is negative. To do this we have to abide by a set of rules that is used in the majority of digital computers on the market today.

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.

$0100\ 1111 = 79_D$ $\qquad$ $1100\ 1111 = -79_D$

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

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:

$$\begin{array}{rll} & 0100\ 1111 & = 79_D \\ + & 1000\ 1101 & = -13_D \\ \hline & 1101\ 1100 & = -92_D \end{array}$$

The answer (-92_D) is obviously wrong ! It should be $+66_D$

Two's Complement representation.

To overcome this anomaly, some brilliant brain, 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.

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_D

```
16 = 0001 0000

     1110 1111   ............One's complement

             1   ............Add 1
     ----------
     1111 0000   ............2's complement.
```

Find the 2's complement of 102_D

```
102 = 0110 0110

      1001 1001 ............One's complement

              1 ............Add 1
      ----------
      1001 1010 ............2's complement
```

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

1) NOT number
2) Add 1

let us now use our original example, and add $+79_D$ to -13_D we will see that this method works correctly.

NOT $13_D = -14_D$. $-14_D = 1111\ 0010_B$

1111 0010 + 1 = 1111 0011 2's complement

```
    0100 1111   = +79D
+   1111 0011   = -13 2's complement
------------
    0100 0010   = +66D
```

+66 is the correct answer !

Add (-3) to (-2)

```
              3D  = 0000 0011B    2D  = 0000 0010B
complement ..... 1111 1100          1111 1101
 add 1     .....         1                   1
                 ---------           ---------
                 1111 1101           1111 1110
```

```
      -2  = 1111 1110 2's complement
+
      -3 =  1111 1101 2's complement
      ---------------
      (1)  1111 1011
```

The carry from **bit 7** is ignored completely, and our result is 1111 1011 which should be the 2's complement of -5. We can test this quite easily by working in reverse.

```
NOT 1111 1011  = 0000 0100
                         1 .......add 1
                 ---------
                 0000 0101  = 5D
```

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

The range of numbers that can be stored in a 8-bit 2's complement representation is $+127_D$ down to -128_D. (see table 2.7)

Table 2.7 2'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
14	0000 1110	-14	1111 0010
9	0000 1001	-9	1111 0111
8	0000 1000	-8	1111 1000
7	0000 0111	-7	1111 1001
6	0000 0110	-6	1111 1010
3	0000 0011	-3	1111 1101
2	0000 0010	-2	1111 1110
1	0000 0001	-1	1111 1111
0	0000 0000		

Even though we have only discussed 8-bit integers, everything we have said also applies to 16-bit integers. The number range for unsigned 16-bit integers is 0 - 65535_D, and for 16-bit 2'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.

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 chapter at a later stage, and a lot of it will sink in automatically as you progress through this book.

Chapter Three

Strings & Things

In the previous chapter we discussed the use and storage of numeric variables. Well, in this chapter we are going to talk about another type of variable, the **STRING VARIABLE.**

The Amstrad can not only manipulate numerical values, but is a master at juggling with letters and text. **Strings** are simply strings 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 **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange. 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.

Ascii is coded into 7 bits (0 to 6), and we have already seen in the last chapter that 7 bits can hold a number in the range **0 to 127_D 0000 0000 to 0111 1111_B**. This suggests that **128** different codes can be defined from one byte, and these 128 characters form the **standard Ascii character set**. Graphic characters are allowed by setting bit 7 which allows a further 128 characters to be defined 128 to 255_D , 1000 0000 - 1111 1111_B.

Table 3.1

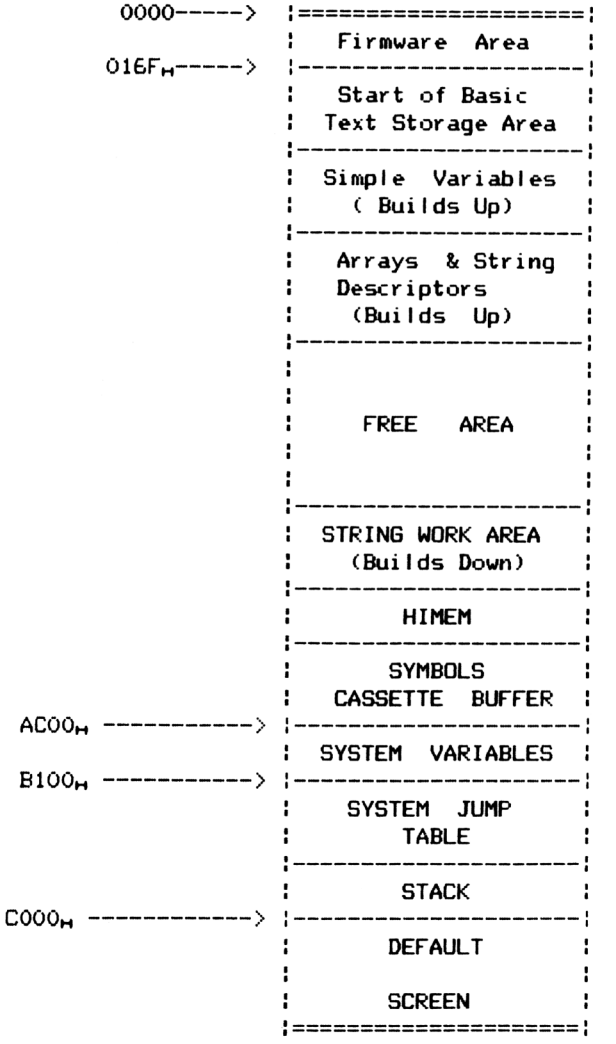
THE ASCII CHARACTER CODES

CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER
0		43	+	86	V
1		44	'	87	W
2	Cursor off	45	-	88	X
3	Cursor on	46	.	89	Y
4		47	/	90	Z
5		48	0	91	[
6		49	1	92	\
7	Sound Bell	50	2	93	]
8	Backspace/Erase	51	3	94	
9	Cursor ==>	52	4	95	_
10	Cursor Down	53	5	96	'
11	Cursor Up	54	6	97	a
12	Home Cursor\Cls	55	7	98	b
13	Carriage Return	56	8	99	c
14		57	9	100	d
15		58	:	101	e
16		59	;	102	f
17		60	<	103	g
18	Erase to E.O.L	61	=	104	h
19		62	>	105	i
20	Erase to E.O.W	63	?	106	j
21		64	@	107	k
22		65	A	108	l
23		66	B	109	m
24		67	C	110	n
25		68	D	111	o
26		69	E	112	p
27	ESC	70	F	113	q
28		71	G	114	r
29		72	H	115	s
30	Home	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		

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 form 0 to 31, known as **Control Codes**, differ from computer to computer. **See Table 3.1**

Diagram 3.1

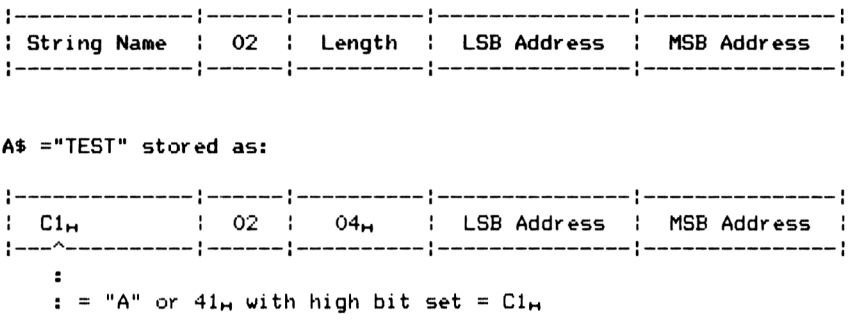
MEMORY ALLOCATION



String Variables store their data in the actual Basic line, unless they have been changed in some way after being defined, in this case they are moved into the **String Work Area** , and a pointer to their address is stored in the **string descriptor block,**the way this data is stored is shown in Dia 4.1. The **default starting address for Basic is $016F_H$**, this is where Basic lines are stored as you type them in from the keyboard. The number of bytes used by a string is dependant on the length of the string, in characters, and the length of the string name; each character in the string uses one byte as does each character in the string name.

Dia 3.2

String Descriptor Block



String variables cannot be operated on in the same manner as numeric variables - after all, who would want to use an expression, Y$ = “KEITH”:X$ = “YOU”:IF Y$/X$ = “HELLO” THEN GOTO .....? 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'm You"

20 Y$ ="r New Am" : Z$ ="strad"

30 PR$ = W$+X$+Y$+Z$

40 PRINT PR$                    ===> I'm Your New Amstrad
```

Strings may be compared in the same manner as numeric 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
```

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 X$=INKEY$

30 IF X$="" THEN GOTO 20

40 LOCATE 1,1:PRINT X$;" = ";ASC(X$);" ASCII CODE"

50 GOTO 20
```

ASC

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

```
10 X$= "A":PRINT ASC(X$)      returns  65D

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

When comparing two or more strings, Basic compares the Ascii value of each character in the string. For example: AB$ will be greater than AA$, and AC$ will be less than AX$. Also note that "**a**" is greater than "**A**". (See Table 3.1)

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

Earlier, we saw how strings could be concantenated by using the + **sign**, we are not, however, allowed to **TRUNCATE** by using the **- (minus) sign**. We have to **truncate** indirectly by using **LEFT$,RIGHT$,IN$,or MID$** functions. These functions allow us to access part, or the whole of the string, starting in the middle,from the left, or from the right, while the **INSTR** function lets us search for a portion of a string starting from a place specified in the parameters.

LEFT$

This function returns the first **n characters** starting from the left of a string variable.

```
10 A$ ="AMSTRAD"

20 B$=LEFT$(A$,3)

30 PRINT B$

RETURNS B$ = "AMS"
```

RIGHT$

The RIGHT$ function returns the **last n characters** starting from the right of the string variable.

```
10 A$="AMSTRAD"

20 B$=RIGHT$(A$,4)

30 PRINT B$
```

Returns B$ "STRAD".

MID$

MID$ is used to take part of a string of length **n** starting at position **p**.

```
10  A$="THIS DEMONSTRATES THE MID$ FUNCTION"

20 B$=MID$(A$,6,12)

30 PRINT B$
```

Returns B$ "DEMONSTRATES"

The MID$ function can also be used on the **right side** of the argument to modify a specified 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"

INSTR

This function allows a string to be searched for any occurance of another string. If the substring does exist **INSTR returns the starting position** of the substring.

When using this function it should be noted that the **whole substring must be contained in the search string**, or the function will return a zero.

```
LET A$="ABCDEFG"

INSTR(A$,"EFG")          result returned = 5

INSTR(A$,"FGH")          result returned = 0

INSTR(2,A$,"ABCD")       result returned = 0

INSTR(2,A$,"DEF")        result returned = 4
```

The last example searched for the substring "DEF" starting from the second position in the search string and returned a value that informs us substring "DEF" starts at position 4 in the search string.

CHR$

In an earlier part of this chapter we used the **ASC** function to convert from a character to it's equivalent **Ascii code**. The **CHR$** function lets us convert in the opposite direction: it converts **from Ascii code to a character.**

CHR$ permits us to use **unprintable characters** within our programs, and it is an extremely powerful function.

"Why do we want to print the unprintable ?" I hear you ask.

Have you no 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 different directions on the screen. Or, tell me, how do you print the sound of a bell ? With CHR$(7), of course ! Go on, try it.

```
10 CLS:LOCATE 1,24:PRINT "XXXXXXXXXXXXXXXXXXXXXXXXX
XXXXX";
20 PRINT CHR$(7)
30 FOR I=1 TO 100:NEXT
40 GOTO 20
```

See what I mean ? You have just printed your first unprintable character, and the CHR$ function allows us to do some clever things in our programs. Line 10 is there to show that even though it is sounding the bell, it is actually printing to the screen.

CHR$(n) can also be used to compare single character strings:

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

You can even use it in tandem with another function. Try to work through the following program.

```
10 X$="A"

20 INPUT A$

30 IF A$ = CHR$(ASC(X$)) THEN PRINT "YES" ELSE

PRINT "NO"

40 GOTO 40
```

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:LINE INPUT A$

20 FOR I = 1 TO LEN(A$):X=X+1

30 NEXT

40 PRINT "A$ IS";X;" CHARACTERS IN LENGTH"

50 GOTO 10
```

LOWER$

LOWER$ will turn any upper case characters (capital letters) into lower case characters.

```
10 LINE INPUT X$
20 FOR I = 1 TO LEN(X$)
30 IF MID$(X$,I,1)=CHR$(30) OR MID$(X$,I,1)=32
OR I = 1 THEN GOTO 50
40 MID$(X$,I,1)=LOWER$(MID$(X$,I,1))
50 NEXT I
60 GOTO 10
```

UPPER$

This function performs in the same way as LOWER$ but in reverse.

HEX$** and **BIN$ are useful for converting from one number system to another, and could be used within a program that converts between number bases.

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

Two functions we haven't yet mentioned are **SPACE$** and **STRING$.** Both of these statements are valuable aids when padding out a string, or filling in a display with fill characters.

With STRING$ you can create a string of identical characters.

```
10 X$=CHR$(129)+CHR$(132)
20 PRINT STRING$(40,X$)
```

```
10 FOR I = 1 TO 24
20 PRINT STRING$(40,140)
30 NEXT
```

Did you notice how quickly the screen filled up with characters ?

Finally, we have two really powerful string functions: **VAL** and **STR$.**

VAL converts a string variable, or expression into a numeric expression represented by the characters in a string argument - in fact, it e**VAL**uates it. The string must be numeric, and if it is a **real** number it must contain a decimal point or exponent (**E**).

```
B$ ="100.60" :PRINT VAL(B$)      ======> returns 100.60
B$="999999999":PRINT VAL(B$)     ======> returns 999999999
B$="9999999999":PRINT VAL(B$)    =======> returns 1E10
B$="ABC45": PRINT VAL(B$)        =======> returns 0
B$="45ABC64":PRINT VAL(B$)       =======> returns 45
B$="1E-3":PRINT VAL(B$)          =======> returns 0.001
```

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

```
10 A$=INKEY$

IF A$="" THEN GOTO 10

30 X=VAL(A$)

40 IF X<1 OR X>9 THEN GOTO 10

50 PRINT  "YOU PRESSED A NUMBER BETWEEN 1 & 9"

60 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 be edited and turned back into a numeric value with VAL, or printed to the screen/printer, and printed in a predetermined format.

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**. Also, PRINT A prints the value with a **trailing blank**, and PRINT STR$(A) prints the value **without a trailing blank.**

```
A = 1650:PRINT LEN(STR$(A))      will return 5

A = -1650:PRINT LEN(STR$(A))     will return 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 LOCATE command. For instance, how many times have you seen, or used, yourself, Basic code that resembles the following:

```
10 LOCATE 3,5:INPUT "CHOOSE A NUMBER BETWEEN

1 & 10";X

20 IF X>10 OR X<1 THEN LOCATE 3,5:PRINT "      ";

30 LOCATE 3,5:PRINT "ILLEGAL INPUT";

40 FOR I = 1 TO 100:NEXT

50 LOCATE 3,5:PRINT "              ";

60 GOTO 10
```

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

```
10 LOCATE 3,5:INPUT "CHOOSE A NUMBER BETWEEN
1 & 10";X
20 IF X<1 OR X> 10 THEN PRINT CHR$(11);CHR$(18);
ELSE GOTO 100
30 LOCATE 3,5:PRINT "ILLEGAL INPUT"
40 FOR I=1 TO 100:NEXT
50 PRINT CHR$(11);CHR$(18);:GOTO 10
100 PRINT CHR$(11);CHR$(8);:LOCATE 3,5:PRINT "O.K"
110 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$(11) which moves the cursor up one line (a carriage return followed you pressing the ENTER key),CHR$(18) then clears to the end of the line before printing the next message.

If you study Table 3.1 you will see that control codes support turning the cursor on/off, and you can place it anywhere within a given window simply by using the other control codes.

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

```
10 X$=CHR$(240)+CHR$(10)+CHR$(8)+CHR$(8)+CHR$(242)
+CHR$(9)+CHR$(243)+CHR$(10)+CHR$(8)+CHR$(8)
+CHR$(241)
20 LOCATE 20,20: PRINT X$
```

PSST! DO YOU WANT TO KNOW A SECRET ?

Before we leave the subject of strings and their usefulness when using graphics, I'm going to show something very different that you will not find in books or manuals. If you are a novice don't worry about the commands we haven't discussed, try the program - you **will** understand how it works before you finish the book.

TRY THIS PROGRAM

```
10 CLS
20 X$=STRING$(40," ")
30 PK = @X$
40 AD = PEEK(PK+2)*256+PEEK(PK+1)
50 FOR I = AD TO LEN(X$)+AD
60 POKE I,238
70 NEXT
80 FOR I = 1 TO 24
90 PRINT X$;
100 NEXT
```

The above program brings together important points we have discussed in the previous chapter, and uses some of the string functions from this chapter.

The implications of the above program may not be obvious to you at this moment, but I can assure you, this is a very clever way to point your graphics at strings.

The program starts by setting up a dummy string (X$) with 40 spaces. You can set up any dummy string as long as you use a string with a length equal to the number of characters you are going to store in it. **Line 30 gets the actual address of the string**. The "@**variable name**" will return the address of any variable that is currently initialised under Basic. When used with strings it returns the address of the descriptor block, starting with the address that holds the **string length** (see Dia 3.3). Line 40 allows for this by looking at this address +2 which is the **Msb of the address where the string is stored.** The correct address is calculated by multiplying the Msb by 256 and adding the Lsb - remember Chapter Three ? Line 50 ascertains the length of X$, and Line 60 fills the memory locations, occupied by the string, with graphic character 238 by POKING it into memory. Lines 80 & 90 are added just to prove that it does work.

In the previous example program we formed a string from graphic characters and control codes, but wasn't it long winded to define the string ? Using the method above, you can set your graphic characters and codes into DATA statements, READ them into a variable, and then POKE them into the string. you can even replace one character within the string, or the whole string, if you wanted to. These strings also print to the screen very quickly, and are an asset to a Basic program when speed is essential.

Now, don't go away - we have a few more tricks to discover, but before we start the next chapter I'll let you sit back and digest the last program.

You know, if you really think about it, there is nothing to prevent a machine code subroutine replacing the the graphic characters, and then calling the routine from **AD** (Line 40). Using this method no **Memory command would have to be initialised.** Of course, any routine would have to be written so that it could run in 255 bytes, but a lot of useful routines can be written to do just that.

JUST TO WHET YOUR APPETITE – TRY THIS

```
 10 CLS
 20 X$= STRING$(22," ")
 30 DATA &21,&01,&01,&CD,&75,&BB,&01,&EB,&03
 40 DATA &C5,&3E,&EE,&CD,&5D,&BB,&C1,&0B,&79
 50 DATA &B0,&20,&F4,&C9
 60 PK = @X$
 70 AD = PEEK(PK+2)*256+PEEK(PK+1)
 80 FOR I = AD TO AD+LEN(X$)-1
 90 READ T
100 POKE I,T
110 NEXT I
120 CALL AD
```

Don't worry about understanding this program, it is included to demonstrate that all things (well, nearly all!) are possible with the Amstrad computer.

The DATA lines 30,40, and 50 contain a short machine code routine to fill the screen with a graphics character. The data represents the actual machine code. AD contains the starting address of the actual string data, so when you **CALL AD** you are calling the machine code routine. This routine can be saved along with the basic program, it requires no setting of **MEMORY**, and Basic will never overwrite it.

Chapter Four

ARRAY! ARRAY!

During the course of this chapter we shall be talking about the **READ,DATA, and DIM,**statements and how they are organised under Locomotive Basic. At the end of the chapter we shall look at another far more powerful way of utilising arrays.

One of the most common uses of the 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 - a shopping list, 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

50 DATA 3,456,255,8,1064,34,10111000,-1,NUMBERS
```

To make use of our **data statements** we use the **READ** command which tells the computer to **read ONE** value

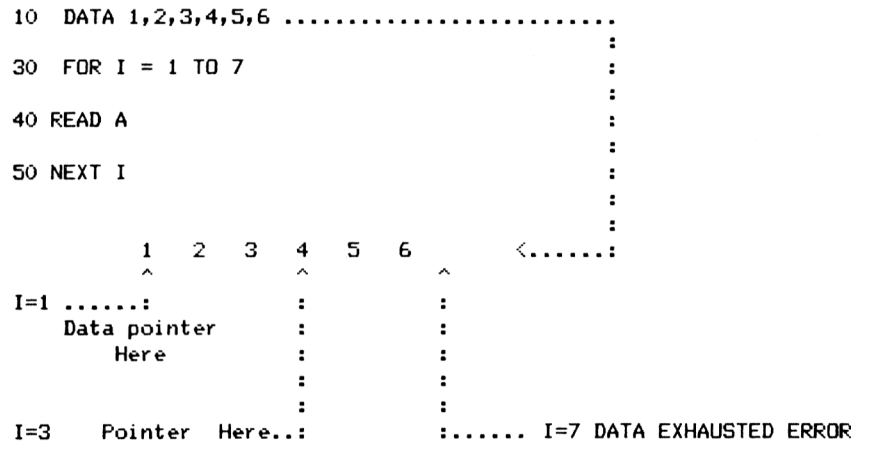
from a data list - **DATA & READ** are always used together. If we add the following lines to the program above, we can get a good idea of how the two statements operate.

```
50  READ A$
60  IF A$ = "X" THEN GOTO 90
70  PRINT A$
80  GOTO 50
90  READ A
100 IF A = -1 THEN GOTO 130
110 PRINT A
120 GOTO 90
130 READ A$
140 PRINT A$
150 END
```

Line 50 tells the computer to read an item from the data list. **In line 60 the program checks to see if A$ = "X" - "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$ is equal to "X", program flow is

directed to line 90 where the same **read** process is repeated but this time using the 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 the items 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.



The above program, when run, results in a **Data Exhausted in 10** error message. This is because the program tried to **read** more items than there were in the **data list**. If we add **line 60 RESTORE 10** and re-run the program, the error condition does not arise because we have told the Amstrad to **restore the data pointer to the very first data item in line 10. RESTORE n** will reset the pointer to the beginning of a specific **Data Line**, and the command is useful when we need to access the data in the list more than once.

RESTORE 10

The above statement would reset the **Data Pointer** to the beginning of the data in Line 10 and the first **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 seen,data types can be mixed (strings and numeric data).

More than one **Data Item** can be read from a **Data List** by a single **READ** statement, and no restrictions are place 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 A,B,C,D,E
30  DATA D,E,",",F
40  READ X,Y,Z
50  END
```

One 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  KEITH,0282,57427,ALBERT,0256,37754,
PHOENIX,01,24356
20  DATA  GORDON,574,69812,BOOTS,0652,57341,
DOCTOR,00,57977
30 CLS
40 FOR I = 1 TO 6
50 READ N$,C,T
60 PRINT N$; "TELEPHONE ";C;"-";T
70 NEXT
```

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

We are now ready to probe an area of computing that most novice programmers find difficult to comprehend.....ARRAYS!

Arrays are the most powerful data structures we have available when programming in Basic. An array is basically (pardon the pun!) 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 easily be 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 mentioned, string or numerical, but are followed by a **subscript** which is enclosed in brackets.

A (9)
Variable..: :.....Subscript

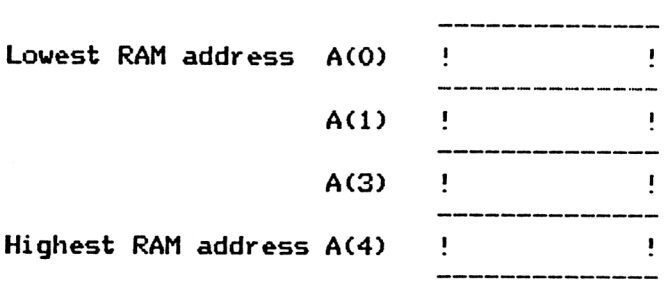
The number, in brackets, is the label to one of the **elements in** the array, and the brackets tell the computer that the variable is a subscripted variable. Thus you can store, and retrieve, data from one specific element in an array by telling the computer which variable, and which subscript.

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 for the CPC 464, 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 the data can easily be modified or changed.

The computer stores arrays in Ram in a consecutive manner as shown in Dia 4.1

Dia 4.1



ARRAY A is stored in memory in exactly the same order as it's subscripts, with A(0) being the lowest memory address, and A(4) occupying the highest memory location.

```
10 CLS

20 FOR I = 0 TO 3

30 A(I) = I

40 NEXT

50 FOR I = 1 TO 3

60 PRINT " THIS IS A(";A(I);")"

70 NEXT
```

This program will print out each element of array **A** with it's correct subscript value as stored in memory.

Another way to enter value into an array is to use the **READ and DATA** statements.

```
10 CLS

20 FOR I = 0 TO 10

30 READ A(I)

40 NEXT

50 INPUT X

60 PRINT A(X)

70 GOTO 60

80 DATA 365,400,1,67,32,56,899,3.33,40000
```

The above program demonstrates how it is possible to access data that is 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, notice 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),say, would result in a **B.S ERROR (Bad Subscript)**, error message - this is the Amstrad's way of telling us that we have tried to access an element that didn't exist.

To create an array of more than 11 elements (subscripts 0 - 10), we must use the **DIM** statement. **DIM A(20)** tells the computer to reserve 21 locations (0-20) for the storage of the 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,TREFOR,GEOFF,HARRY,PETER,
NORMAN
510 DATA PAT,MIKE,SHIRLEY,KATHY,COLIN
```

The above program provides a way of creating a list of all our friends. By placing a value at the very beginning of the data, we can easily add data without having to edit the main part of the program - each time we add a new friend's name, 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)
500 4,KEITH, 20 Chapel St,Burnley,BARRY,
4a Ascough Close,Burnley
510 GEOFF,36 Peyton Place,Amsham,HARRY,
67 Darlington Ave,Fictown
```

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

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

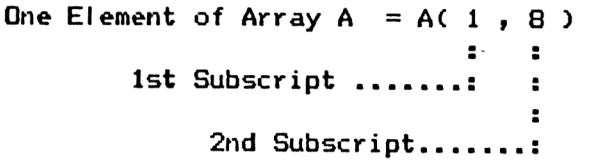
Line 140 makes I = B because we jumped out of the FOR NEXT loop before the loop had terminated. Although it isn't necessary to do this, it is good programming practice to terminate FOR NEXT loops.

From the above demonstration you can see that **strings 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.

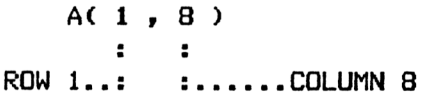
Encounters of the second kind.

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

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



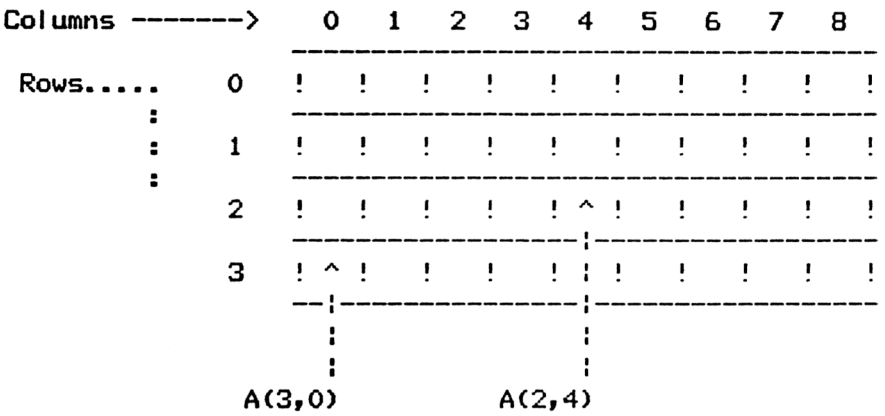
A two-dimensional array can be visualised as a table made up of **Rows & Columns.** In general, the first subscript is the Rows and the second subscript the Columns.



DIM A(3,8) tells the computer to reserve enough space in Ram for an array 4 columns by 9 rows = 36 storage boxes.

```
10 DIM A(3,8)

20 FOR ROW = 0 TO 3

40 FOR COL = 0 TO 8

40 A(ROW,COL) =COL+ROW

50 NEXT COL

60 NEXT ROW
```

Dia 4.2



A Two-dimensional Array

From Dia 4.2 you can see that whenever we want to pick a particular location within the array, we can refer to it by it's **Row & Column** numbers.

TO **READ** values into a two-dimensional array we have to use a **NESTED** FOR,NEXT LOOP, and we need to set one of the loops equal to the number of Rows, and one equal to the number of columns.

The program 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 value of Col (0), and COL is then incremented by one, and A(0,1) is filled with value of COL (1). When COL is equal to 8, ROW is incremented by one, and the next Row is filled until COL = 8 again, and the pattern is repeated until ROW = 3. Let's prove this by re-writing the above program to print out the values in each memory location to the screen.

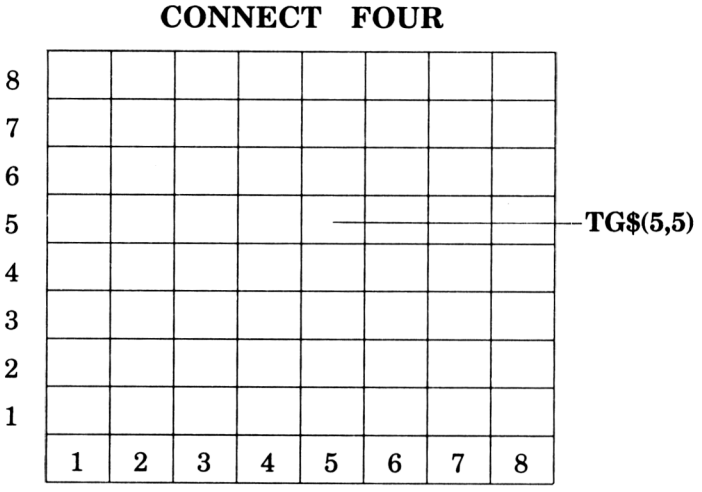
LISTING TWO

```
10 CLS

20 DIM A(3,8)

30 FOR ROW = 0 TO 3

40 FOR COL = 0 TO 8

50 A(ROW,COL) =COL

60 NEXT COL

70 NEXT ROW

80 FOR ROW = 0 TO 3: PRINT "Row ";ROW;"= "

90 FOR COL = 0 TO 8

100  PRINT A(ROW,COL);

110  NEXT COL

120  PRINT

130 NEXT ROW
```

Listing two provides us with a good 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 are familiar with the general rules. Listing two follows the original game very closely with the Amstrad taking the role of your opponent - it plays a pretty good game !

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

Dia 4.3



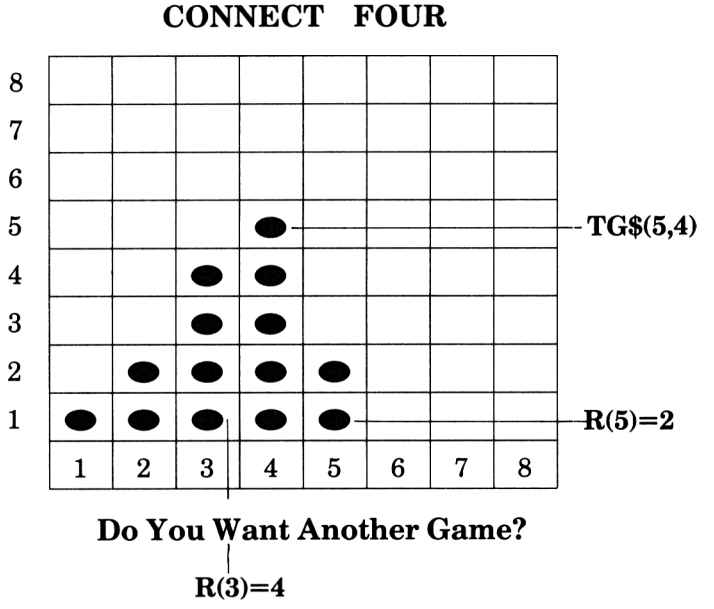
From Dia 4.3 it is obvious that the Connect Four board is very well suited to a two-dimensional array TG$(8,8). You will notice within the program we have not used TG$(0,0), it still exists in memory but it is our perogative not to use it if we so choose.

In the game of Connect Four the counters are placed on the board in a bottom to top manner - this is why the rows are numbered 8 to 1 (Dia 4.4). The row data is read into DN(8) array with the highest Y position first so that DN(1) holds the Y position for the 20th line on the screen, and DN(2) holds the Y position for line 18 on the screen. The X position is calculated by multiplying the player's choice of column by 3, and adding 7 to **LOCATE** the cursor at the correct print position.

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's counter.

Array R(8) is used to keep track of how many pieces are occupying positions in a particular column.

Dia 4.4



The array G(16) holds the **evaluation functions** that are used by the computer when deciding which move to make. You can experiment with them by trying different values, and watching how the changes affect play.

If you really want to analyse the program, now is a good time to try the **TRON** function. Switch it on and follow the program by looking at the listing and checking program flow as the line numbers are printed on the screen. You can even stop the program by pressing **ESC ESC**,and examining some of the variables.

LISTING 2

```
5 DEFINT A-C,D,H,I,J-T,V-Z
10 DIM TG$(8,8),A(4),R(8),K(4),J(4),G(16),DN(8),TH$(8)
20 REM             AMSTRAD CPC 464
30 REM  "CONNECT 4" FOR COMPUTER AND ONE PLAYER
40 MODE 1:GOSUB 390:GOSUB 1130:BORDER 0:INK 0,0:INK 1,26:INK 2,21:INK 3,6
50 LET TH$=STRING$(8,CHR$(230))
60 MODE 1:GOSUB 470:GOSUB 280
70 GOTO 540
80 REM ********** MAIN CALCULATING **********
90 SOUND 1,100,8
100 O$=H$:IF X$=H$ THEN O$=C$
110 Y=1:YY=0:S=0:GOSUB 160
120 Y=1:YY=1:GOSUB 160
130 Y=0:YY=1:GOSUB 160
140 Y=-1:YY=1:GOSUB 160
150 RETURN
160 XX=1:A=1:Q=0:S=S+1
170 M=0
180 FOR I=1 TO 3:H=X+I*YY:N=R+I*Y
190 IF H<1 OR N<1 OR H>8 OR N>8 THEN GOTO 250
200 G$=TG$(N,H):IF M=0 THEN 230
210 IF G$=O$ THEN I=3:GOTO 260
220 Q=Q+1:GOTO 250
230 IF G$=X$ THEN A=A+1:GOTO 250
240 M=1:GOTO 210
250 NEXT I
260 IF XX=0 THEN A(S)=A:K(S)=Q:RETURN
```

```
270 XX=0:YY=-YY:Y=-Y:GOTO 170
280 REM ********* DRAW BOARD *********
290 PEN 2:PAPER 3:J=49:FOR I=10 TO 32 STEP 3:LOCATE I,22:PRINT CHR$(J)+" ":J=J+1:NEXT
300 PAPER 0
310 FOR I=136 TO 520 STEP 48:PLOT I,40:DRAW I,328:NEXT
320 FOR I=40 TO 328 STEP 32:PLOT 136,I:DRAW 520,I:NEXT
330 LOCATE 15,2:PEN 3:PRINT"CONNECT FOUR":PEN 1:RETURN
340 REM ********* MAIN INPUT ROUTINE *********
350 LOCATE 10,25:PRINT SPACE$(30);:PEN 2:LOCATE 10,25:PRINT MES$;
360 IN$=INKEY$:IF IN$="" THEN 360
370 RETURN
380 REM ********* DEFINE CHARACTERS **********
390 SYMBOL AFTER 90:FOR I=93 TO 96:READ N1,N2,N3,N4,N5,N6,N7,N8
400 SYMBOL I,N1,N2,N3,N4,N5,N6,N7,N8:NEXT
410 DATA 7,31,57,127,126,59,28,7,224,248,156,254,126,220,56,224
420 DATA 3,15,153,185,255,191,152,7,192,240,153,157,255,253,25,224
430 H$=CHR$(93)+CHR$(94)
440 C$=CHR$(95)+CHR$(96)
450 RETURN
460 REM ********* INITIALISE VARIABLES *****
470 DATA 20,18,16,14,12,10,8,6
480 FOR I=1 TO 8:READ DN(I):NEXT
490 REM ******** VALUES BELOW CONTROL COMPUTER EVALUATIONS ********
500 REM ******** TRY CHANGING THEM AND SEE HOW IT EFFECTS PLAY ***********
510 DATA 1,120,505,1E22,1,880,3000,1E30,1,80,1000,1E16,1,475,3050,1E14
520 FOR I=1 TO 16:READ G(I):NEXT
530 RETURN
```

```
540 SOUND 1,100,15:SOUND 0,50,7:MES$="Do you want to go first?":GOSUB 350
550 IF IN$="Y" OR IN$="y" THEN 580
560 IF IN$<>"N" AND IN$<>"n" THEN 540 ELSE X=INT(RND*8)+1:GOTO 990
570 REM ******** HUMAN ROUTINE *******
580 PRINT CHR$(7):LET MES$="Pick a number (1 to 8)":GOSUB 350
590 X=INT(VAL(IN$)):IF X=0 THEN 580
600 IF X>=1 AND X<=8 THEN 620
610 GOTO 580
620 R=R(X):IF R>7 THEN 580
630 R(X)=R+1:R=R+1:E=X*3+7:F=DN(R):TG$(R,X)=H$
640 LOCATE E,F:PEN 2:PRINT H$;
650 X$=H$:SOUND 1,100,7:GOSUB 90
660 FOR I=1 TO 4:IF A(I)<4 THEN NEXT:GOTO 710
670 I=4
680 INK 1,6,0:PEN 1:LOCATE 10,25:PRINT SPACE$(30):LOCATE 12,25:PRINT"<<< OK YOU WIN! >>>"
690 FOR BB=1 TO 4000:NEXT::INK 1,26:GOTO 1120
700 REM ******** COMPUTER THINKING *******
710 P6=0:MES$="Thinking....":LOCATE 10,25:PRINT SPACE$(30)
720 LOCATE 10,25:PRINT MES$;:Z1=23:Z2=25:LOCATE Z1,Z2:PEN 2:PRINT TH$;
730 U=0:J=1:FOR P=1 TO 8:R=R(P)+1
740 IF R>8 THEN 940
750 E=1:X$=C$:F=0:X=P
760 GOSUB 90
770 FOR L=1 TO 4:J(L)=0:NEXT
780 FOR I=1 TO 4:A=A(I):IF A-F>3 THEN I=4:GOTO 990
790 Q=A+K(I):IF Q<4 THEN GOTO 810
800 E=E+4:J(A)=J(A)+1
```

```
810 NEXT I
820 FOR I=1 TO 4:W=J(I)-1:IF W=-1 THEN 840
830 Z=8*F+4*SGN(W)+I:E=E+G(Z)+W*G(8*F+I)
840 NEXT I
850 IF F=1 THEN 870
860 F=1:X$=H$:GOTO 760
870 R=R+1:IF R>8 THEN 900
880 GOSUB 90
890 FOR I=1 TO 4:IF A>3 THEN E=2 ELSE NEXT I
900 IF E<U THEN 940
910 IF E>U THEN O=1:GOTO 930
920 O=O+1:IF RND>1/O THEN 940
930 U=E:P6=P
940 LOCATE Z1,Z2:PEN 3:PRINT CHR$(231);:Z1=Z1+1:SOUND 1,500,14:NEXT P
950 IF P6<>0 THEN 980:ELSE LOCATE 10,22:PEN 1:PRINT CHR$(5);
960 LOCATE 5,25:PRINT"~~~~~It's a draw~~~~~"
970 FOR a=1 TO 1000:NEXT:GOTO 1120
980 X=P6
990 LOCATE 10,25:PEN 1:PRINT"I'm going in column";X;"       ";
1000 R(X)=R(X)+1:R=R(X)
1010 TG$(R,X)=C$
1020 X$=C$
1030 E=X*3+7:F=DN(R):SOUND 0,100,23,15
1040 FOR I=1 TO 3:LOCATE E,F:PRINT"  ";:FOR BB=1 TO 100:NEXT BB:LOCATE E,F
1050 PEN 3:PRINT C$;:FOR BB=1 TO 100:NEXT BB:NEXT I:SOUND 1,17,23,8
1060 GOSUB 90
1070 FOR I=1 TO 4:IF A(I)<4 THEN NEXT I:GOTO 580
```

```
1080 I=4
1090 LOCATE 5,25
1100 FOR I=1 TO 8:PEN 2:LOCATE 10,25:PRINT"<<<< SORRY I WIN >>>>";:FOR BB=1 TO 500:NEXT BB
1110 LOCATE 10,25:PEN 3:PRINT"\\\\ HA! HA! HA! ////":FOR BB=1 TO 500:NEXT BB:NEXT I
1120 FOR a=1 TO 100:NEXT:RUN
1130 BORDER 13:INK 0,13:INK 1,0:INK 2,24:INK 3,26:CLS
1140 CLS:PEN 1:LOCATE 14,1:PRINT"INSTRUCTIONS":LOCATE 14,2:PEN 3:PRINT STRING$(12,CHR$(208))
1150 PRINT:PEN 2:PRINT"CONNECT FOUR";:PEN 1:PRINT":"
1160 REM    N.B.  1 is control/0
1170 PRINT:PRINT"   The  game  consists of  placing  your markers on   a  board  with  the
intention of trying to get  four of your
markers in a row."
1180 PRINT:PRINT"   Either:                                                Diagonally
           Vertically
            or Horizontally"
1190 PRINT:PRINT" The computer will try to out wit you so be on your guard."
1200 PRINT:PRINT"        Now press "+CHR$(34)+"ENTER"+CHR$(34)+" to start."
1210 IF INKEY$=CHR$(13) THEN RETURN ELSE 1210
```

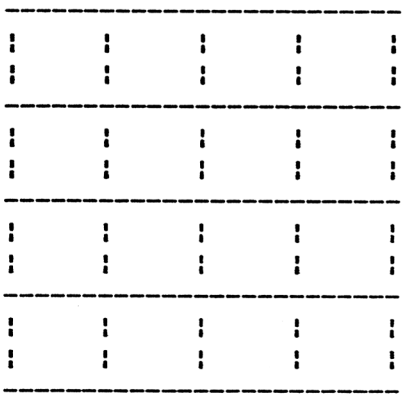
BEYOND THE THIRD DIMENSION

We have now reached the point where the faint hearted give up and fall by the way-side - they lose the ability to come to terms with the extra dimensions. Don't be one of the majority, take some time to understand arrays, and you will soon realise multi-dimensional arrays are very useful programming tools that can be used to advantage, and are not as difficult to work with as they are believed to be.

As an example, consider the case of a friend of mine, Stephen Crew. Steve is a hardware merchant and runs a very successful retail business,**S.CREW IRON-MONGERS LTD**.

As you can imagine, there are hundreds of small, different items that are stocked by an 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 to 8 inches in 1/16th of inch increments. Steve is a tidy sort of chap and he keeps the screws in drawers similar to Dia 4.5

Dia 4.5



Stephen is a logical thinking sort of fellow,and it wasn't long after he purchased an Amstrad computer that he realised he could write a program to keep a record of all his screws by simply using an array to hold all the data. His screws were kept in two cabinets each holding 4 drawers, and each draw contained every size up to 1 inch increments so that drawer one had screws from 1/16th of an inch to 1 inch. Drawer two had screws from 1 1/16th of inch to 2 inches, and so on up to drawer eight which had screws from 7 1/16th of an inch up to 8 inches.

Steve had never used multi-dimensional arrays, but this was how he started his program.

```
SCW (4, 4, 4, 2)
     :  :  :  :....Cabinet
     : Col :
     :     :.Draw
 Row.:
```

```
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 STK
70 SCW(ROW,COL,DRAW,CAB)=STK
80 NEXT COL
```

```
 90 NEXT ROW

100 NEXT DRAW

110 NEXT CAB



500 DATA 3600,2000,4000,12000,1200,4560,2389,1907

510 DATA 1200,1324,5279,567,1290,4317,2345,6732

520 DATA ...........
```

Once he had filled the array with the stock figures, Steve knew that he could find the amount of stock by accessing the correct element of the array.

```
Element SCW(1,1,1,1) held the 1/16th screws.

Element SCW(1,1,2,1) held the 1 1/16th screws.

Element SCW(1,1,4,2) held the 7 1/16th screws.
```

ANOTHER AMSTRAD SECRET

In the last chapter we discussed, and indeed proved, that a machine code program could be loaded into a string and then called from Basic. One draw-back with this method is that parameters have to be passed by defining a specific memory location to allow the routine to grab' them e.g. by **Poking** the parameter, and **Peeking** the result. You have witnessed how easy it is to use arrays once the initial ground rules have been absorbed, imagine then, how easy it would be to pass arguments to a machine code subroutine by simply loading an element of an array with the argument : A(1,3) = &FF.

Type in the following program and run it.

```
10   DEFINT A-Z :DIM A(12)     ' Working in
integers only
20   FOR I = 0 TO 12
30   READ A(I)
40   NEXT
50   A(1) = 8
60   A(4) = 16
70   GOSUB 1000
80   A(1) = 7
90   A(4) = 34
100  GOSUB 1000
110  A(1) = 13
120  A(4) = 8
130  GOSUB 1000
140  A(1) = 12
150  A(4) =  9
160  END
500  DATA &3E00,&0000,&FEF5,&3E07
510  DATA &0000,&2000,&E602,&5F3F
```

```
520  DATA &FEF1,&300E,&4B04,&34CD

530  DATA &C9BD

1000 CALL @A(0):RETURN
```

After running the above program, if you are a novice, you will be very puzzled by it; experienced programmers will realise that this way of initialising machine code subroutines opens up a whole new ball game'.

However, let's start from scratch. The data statements are copies of a machine code routine that by-passes Basic's sound commands and allows us to access the sound registers direct. Don't worry, at this moment, how it works, we shall be dealing with the sound chip in greater detail in a later chapter. Basic lines 50 to 150 deal with passing parameters to the routine, and line 1000 locates the start of the array in memory then runs the machine code section which has been stored in the array ! The advantages of using arrays rather than strings for saving and running machine code routines are twofold: memory is not limited to 255 bytes, which means that larger machine code sections can be used. Parameters can be passed from within Basic quite easily by loading values into the corresponding elements of the array. Also, if a large amount of data is required by the assembly routine, another array can be set up, and the address of this new array can be passed to the machine code array by using the @ **variable** format.

Because this method is novel, there are certain rules that must be obeyed when setting up the array to accept the code. For the impatient, I will now explain how this method works.

The Basic programmer who wishes to return to this subject later, can skip to the next chapter.

Setting up the array:

Listing three is the actual assembly listing used when writing the above routine.

LISTING THREE

```
;************************************************
;  Demonstration of using machine code within a
;  Basic Array then running the array.
;  Assembled using KUMA'S ZEN EDITOR ASSEMBLER.
;************************************************
;
;
;
1   00              NOP             ;alignment
2   3E00            LD A,00
3   00              NOP             ;alignment
4   F5              PUSH AF
5   FE07            CP 07
6   3E00            LD A,00
7   00              NOP             ;aligment
8   00              NOP             ;alignment
9   2002            JR NZ,SKIP
10  E63F            AND 3FH
11  5F      SKIP:   LD E,A
12  F1              POP AF
13  FE0E            CP 0EH
14  3004            JR NC,BACK
15  4B              LD C,E
16  CD34BD          CALL 0BD34H     ;SEND SOUND VAL
17  C9      BACK:   RET
18  00              END
```

When using this method the assembly routine **must be re-locatable.** The reason for this is simple: Basic arrays are stored, in memory, immediately in front of the simple variable list - whenever the variables are changed or

added to the arrays are moved up in memory, which means that a **Call** to another section of code within the routine will no longer call at the correct address. Jumps (JPs) must also be avoided.

After you have assembled your code, work out where you need to insert your arguments and insert NOPS. **Each element of the array will take a two byte value (one word).** Try to arrange for the locations that will receive the arguments from Basic to fall on even numbered bytes within the finished routine. Also, if the length of the code is not divisable by two, insert NOPs until it will.

Object Code from Listing Three

```
Byte No. Object Code

0000    00          NOP <-------Nop inserted for alignment
0001    3E00        LD A,00
0003    00          NOP  <-------Nop  inserted  to  receive  argument.
0004    F5          PUSH AF
0005    FE07        CP 07
0007    3E00        LD A,00
0009    00          NOP<---------NOP  inserted  to  receive  argument.
000A    00          NOP <------NOP inserted to make sure
000B    2002        even length code
000D    E63F
000F    5F
0010    F1
0011    FE0E
0013    3004
0015    4B
0016    CD34BD
0019    C9
```

If the NOPs had not been inserted in bytes 0,3,9,and A, the data would not have aligned after being put into the array. Don't forget that the Object code is put into Data statements by reversing normal Z.80 conventions and putting **MSB first, LSB last.** Consequently the first value for the Basic Data Line is &3E00 - the Z.80 will reverse them into the correct order when Basic loads

them into the array's memory locations. Using the above object code the Data is calculated as: &3E00,&0000,&FEF5 .....&C9BD.

You can check that the code is aligned by **reading** the data into the array, and then typing in the **Direct Mode** PRINT HEX$(A(0)). If you are using Kuma's ZEN invoke the monitor by **CALLING 16384** then disassembling from the memory location returned by PRINT HEX$(
A(0)). The code should exactly match the original Object Code.

One further point, all arrays **must be integer arrays**, otherwise the code will not align as expected. Once you have mastered this technique all other methods of interfacing code seem antiquated, and many possibilities are open to the ingenious programmer: A routine that could SORT a Basic array into ascending order - 1000 8-byte records in 9.6 seconds!

Chapter Five

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 two Basic commands that may have had you puzzled in an earlier chapter, **PEEK & POKE.**

With the aid of **PEEKS and POKES** we can access memory locations directly, and the statements can be used to great advantage in our programming.

Although the following example is little, or no use to us, here is a way, using a **Poke**, to make the Amstrad get it's 'Knickers in a twist'.

```
10 DEFINT A-Z

20 CLS

30 INPUT "CHOOSE A NUMBER ";X

40 PRINT "YOUR NUMBER IS ";X

50 Y = RND(50*2)+1

60 PRINT "I AM GOING TO ADD ";Y; "TO YOUR NUMBER ";X

70 POKE @X,RND(100*2)+1

80 PRINT "THE ANSWER IS ";X
```

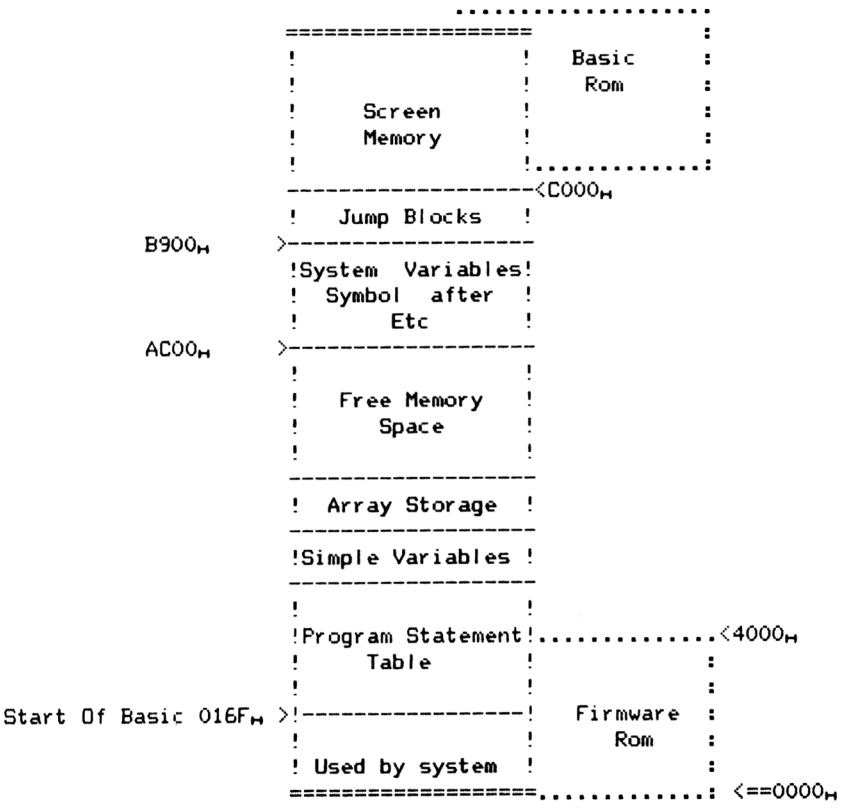
Before we start using **Peeks** and **Pokes** it is necessary to understand the difference between **Rom and Ram**. Also, to utilise the two commands to our advantage we must be aware of what happens within the computer.

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 and 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 each location in turn - 0,1,2....100 etc.

When looking into memory the situation is further complicated by the fact that it is impossible to tell whether we are looking at **Rom** or **Ram,** and it becomes essential to understand which part of memory is **Rom** and which part is **Ram.** We must also be careful of what, and where, we **Poke:** some blocks of memory are used by the system and if we use **Pokes** in a careless manner it is very easy to cause the computer to crash. Although we cannot harm the computer with our **pokeing**, if the system does crash our only recourse is to switch off and start again. This course of action is not at all satisfactory, especially if we have just finished typing a long program into memory, so remember, whenever you use **Pokes** within your program, save it to tape before attempting to run it. If the computer then crashes you have the comfort of knowing that you can reload the program - it saves a lot of typing!

Dia 5.1
Memory Organisation



Organisation of memory is complicated by the fact that the computer has **64K of RAM and 32K of ROM**. The **Z.80 CPU** is only capable of addressing 64K of memory, which means that some method of switching memory must be employed. The CPC464 gets around the problem by splitting the Rom into two sections: **The Firmware Rom and The Basic Rom.**

The **Firmware Rom** is located from 0000 to 4000_Hin the lower part of memory, and is responsible for managing such things as the Programmable Sound generator,PPI. The Basic Rom is at the top of memory from $C000_H$ through to $FFFF_H$ (49152_D - 65535_D). This means that whenever the computer needs to access the **Basic Interpreter** it has to switch out the Screen Ram. However, this switching in and out of memory does not affect the screen display but,further writing to the screen can not take place until the Screen Ram has been re-enabled.

The two Roms may be enabled and disabled separately. You have already been told that we cannot **Poke** into Rom, so it doesn't matter if the Rom is enabled at the location we are **pokeing as no harm will be done to the system.**

System Variables.

Roms can never change their memory contents, and in an ever changing environment such as a Basic program, the computer must have some method of storing various pieces of information, e.g length of Basic program-,current cursor position, top of Basic etc. The Amstrad keeps track of this sort of program information by allocating a block of memory starting at $AC00_H$ (44032_D). This area of memory is very critical and extreme caution should be exercised when pokeing in these regions.

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 tape. The starting address of this table is fixed at **$016F_H$** (367_D) but it's ending address will vary with the size of the program currently in memory. As program lines are added or deleted the end of the PST is altered accordingly.

As each line of 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 $\mathbf{8E_H}$ for the Basic command **DEFINT** because this method saves five bytes of memory. A Basic program has many lines, and most lines have multiple statements, so the saving on memory is vast - as much as 20%. Unfortunately, the **Token Tables** are situated in the Basic Rom so we cannot look at them by using Peeks. Table 5.2 is printed below so you can see how the Tokens relate to the different Keywords.

TABLE 5.2

BASIC TOKENS

KEYWORD	TOKEN	KEYWORD	TOKEN
ABS	FF00	AFTER	80
ASC	FF01	ATN	FF02
AUTO	81	BIN$	FF71
BORDER	82	CALL	83
CAT	84	CHAIN	85
CHAIN MERGE	85 AB	CHR$	FF03
CINT	FF04	CLEAR	86
CLG	87	CLOSEIN	88
CLOSEOUT	89	CLS	8A
CONT	8B	COS	FF05
CREAL	FF06	DATA	8C
DEF FN	8D	DEFINT	8E
DEFREAL	8F	DEFSTR	90
DEG	91	DELETE	92
DIM	93	DRAW	94
DRAWR	95	EDIT	96
ELSE	97	END	98
ENT	99	ENV	9A
EOF	FF4D	ERASE	9B
ERR	FF41	ERL	E3

ERROR	9C	EVERY	9D
EXP	FF07	FIX	FF08
FOR	9E	FRE	FF09
GOSUB	9F	GOTO	A0
HEX$	FF73	HIMEM	FF42
IF	A1	INK	A2
INKEY	FF0A	INKEY$	FF43
INP	FF0B	INPUT	A3
INSTR	FF74	INT	FF0C
JOY	FF0D	KEY	A4
KEY DEF	FF75	LEFT$	DB
LEN	FF0E	LET	A5
LINE INPUT	A6 A3	LIST	A7
LOCATE	A9	LOG	FF0F
LOG10	FF10	LOWER$	FF11
MAX	FF76	MEMORY	AA
MERGE	AB	MID$	AC
MIN	FF77	MODE	AD
MOVE	AE	MOVER	AF
NEW	B1	NEXT	B0
ON GOSUB	B220	ON GOTO	B2A0
ON BREAK GOSUB	B39F	ON BREAK STOP	B3CE
ON ERROR GOTO	B29C	ON SQ GOSUB	B59F
OPENIN	B6	OPENOUT	B7
ORIGIN	B8	OUT	B9
PAPER	BA	PEEK	FF12
PEN	BB	PI	FF44
PLOT	BC	PLOTR	BD
POKE	BE	POS	FF78
PRINT	BF	RAD	C1
RANDOMIZE	C2	READ	C3
RELEASE	C4	REM	C5
RENUM	C6	RESTORE	C7
RESUME	C8	RETURN	C9
RIGHT$	FF79	RND	FF45
ROUND	FF7A	RUN	CA
SAVE	CB	SGN	FF14
SIN	FF15	SOUND	CC
SPACE$	FF16	SPEED INK	CD A2
SPEED KEY	CD A4	SPEED WRITE	CD D9
SQ	FF17	SQR	FF18
STOP	CE	STR$	FF19
STRING$	FF73	SYMBOL	CF

SYMBOL AFTER	CF 80	TAG	D0
TAGOFF	D1	TAN	FF1A
TEST	FF7C	TESTSR	FF7D
TIME	FF46	TROFF	D2
TRON	D3	UNT	FF1B
UPPER$	FF1C	VAL	FF1D
VPOS	FF7F	WAIT	D4
WEND	D5	WHILE	D6
WIDTH	D7	WINDOW	D8
WINDOW SWOP	D8 E7	WRITE	D9
XPOS	FF47	YPOS	FF48
ZONE	DA	=	EF
<	F2	+	F4
-	F5	*	F6
/	F7		

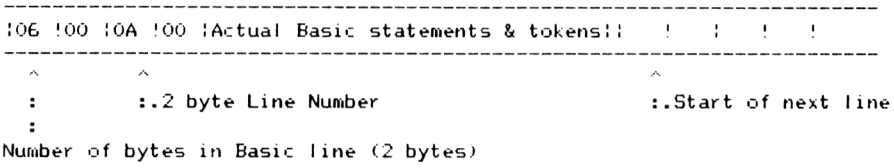
==========

As soon as a 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 the routine that deals with the command or function. When the computer spots a Token, eg. GOTO, it knows immediately which routine and the location of the routine in memory. The secret lies in the way the Tokens are allocated to the various Basic statements.

The Basic Rom contains a **Verb Action List** which holds the addresses for all the routines used by the various Basic commands. To find the correct address the computer has to calculate the displacement into the list according to the value of the Token. The very first Token is $\mathbf{80_H}$ **(AFTER),** and all other Tokens are increments of this number. For example, **Closeout** has a Token value of $\mathbf{89_H}$, and to find the address of the Closeout routine the computer deducts $\mathbf{80_H}$ from the token and adds this to the start address of the Verb Action List. The Execution driver now extracts the address from this location and jumps to the actual routine that will carry out the **Closeout** command.

Following on from the Program Statement Table is the **Variable List Table** (VLT). This table contains the names and values for all variables currently initialised with the Basic program. The list is divided into three sections: **Simple variable names,String Pointers, Subscripted variables.** This table also alters as variables are declared or deleted.

Structure of a Basic Line as represented in memory



The **Variable Type Table (VTT)** is stored as a fixed length table starting at $AE0C_H$ (44556_D). This table is 26 bytes in length with each byte corresponding to a letter of the alphabet (A-Z). At switch on the VTT is initialised by Basic to 05_H in each of the twenty six locations. The 05_H markers denote **Real** values which is Basic's default setting. If a Basic program starts with a DEFINT statement,any variable included within the statement will have the 05_H markers replaced by the integer marker 02_H. A Basic statement: **X!= 33.3333** is accepted because the type marker follows the variable name. However, if the interpreter comes across the statement **Z = 765.8993** Basic checks the VTT to see if the entry under **Z** corresponds to the Real marker (05_H).

Now is as good a time as any to try out a few Peeks in the Amstrad's secrets. The following program will allow you to take a look at the VTT, and you will see how the entries alter for each type of variable.

```
10   CLS
20   GOSUB 150
30   CLS
40  DEFINT A-Z
50  GOSUB 150
60  CLS
70  DEFSTR
80  GOSUB 150
90  CLS
100 DEFREAL
110 GOSUB 150
120 END
150 FOR I%=&AE0C TO &AE0C+25
160 PRINT PEEK(I%);
170 NEXT
180 INPUT "PRESS 'Enter' FOR NEXT PEEK";
190 RETURN
```

We can use Pokes to put our machine code routines into memory. Indeed, we used this method in a previous chapter when we pointed a short routine at a string. Poke can also be used to pass parameters to a machine language routine from within our Basic program. For example, if we had a machine code program to move a blob around the screen, by using a static memory

location to store the current position of the blob, we could Peek the position back to the Basic program, or change the position from Basic by Pokeing the new value into the fixed memory location.

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
70  END
80  DATA 34,56,&45,&65,12 .......
```

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,&CD,ADR3,&F0,ADR4,&45,-1,-1
```

Where ADR1,ADR2 etc represent memory locations. A check for ADR and CODE = -1 is to ensure that the end of data has been reached. The reason CODE is also made to =-1 is simply that it is possible for an address to be -1 as we shall now discover.

Two Rules for PEEK and POKE

In Chapter 3 we unearthed the method of how the Amstrad stores 2-byte numbers: Least Significant Byte First, Most significant Byte Last. Whenever we read a 16-bit (2-byte) value from memory we have to correct for this anomaly by using the following formula:

RULE 1

VALUE = MSB*256 + LSB

OR

PEEK(AD)+PEEK(AD+1)*256 = VALUE

RULE 2

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

POKE ADDRESS = (DECIMAL ADDRESS - 65536)

The reason for this rule is related to unsigned 16-bit integers and the two's complement of a number. (See Chapter 3). In decimal, addresses range from 0 to 65535. However, signed integers, as used by the computer, can only be in the range -32768 to +32767, but the signed integer **-1** = $\mathbf{FFFF_H}$ while the unsigned integer $\mathbf{FFFF_H}$ = $\mathbf{65535_D}$, and this is where the confusion starts. Take a look at the following list.

$32767_D = 7FFF_H$

$32768_D = 8000_H$

$32769_D = 8001_H$

$65535_D = FFFF_H$

$-32768_D = 8000_H$

$-32769_D = 8001_H$

$-65535_D = FFFF_H$

From the above it is clear that all positive integers over 32767 are equivalent to their negative numbers. **X%**, which is an integer variable, will only accept numbers from -32768 to +32767, so to Poke above this limit we must use the negative equivalent. If you don't understand this rule go back and re-read Chapter 3.

Listing 4 will allow you to Peek anywhere in Ram. It will display the memory location followed by whatever is in that location. If the value in memory has an equivalent Ascii value it will be printed, otherwise the program will print the actual value in that memory location. This program is **not** a **disassembler** but it will allow you to investigate a few interesting points of Locomotive Basic.

After typing in the program,look at locations from $AC00_H$ (44032_D) up to $C000_H$. This is the area of memory we discussed a little earlier. Also look at $016F_H$ (367_D), you will see some very interesting characters. You will see the lines of the program you are running.

Unfortunately, because of the way the Roms are structured, you cannot Peek into them - to do this you need a disassembler. A disassembler is a program that automatically disassembles values into their correct Z.80

instructions. Of course, to understand the disassembly you need to have an elementary knowledge of machine code. **Listing 5** is an example of a disassembly. This listing was produced with **Kuma's ZEN** which is also an Editor/Assembler. The first numbers in the listing are the memory locations,and the next values are the actual machine language instructions. The last column is the machine code instructions disassembled into mnemonics.

LISTING 4

```
10  DEFINT A-Z

20  INPUT "STARTING 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 GOTO 40 ELSE GOTO 20

200 IF PK<33 GOTO 250

210 IF PK>122 GOTO 250

220 IF FL = 0 THEN PRINT L;" ";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
```

During the course of the next chapter,as we utilise some of the Amstrad's built in commands, we shall be using machine code programs to illustrate the text. To take advantage of the advance features of the CPC464 a knowledge of assembly language is essential. Teaching this method of programming is beyond the scope of this book but it will be prudent to spend a few paragraphs explaining the rudiments of the language.

```
 MEMORY ADDRESS
 :
 :        ...MACHINE CODE INSTRUCTIONS.
 :        :
ACFE  211010

AD01  CD75BB

AD04  01E803
```

Understanding the above program is difficult. To work out what the program does we would need an index of all the different **Operation Codes** used by the Z.80 CPU. Now take a look at the same program written with an **editor/assembler**.

```
1        ORG  0ACFEH

2  LD HL,1010H

3  CALL 0BB75H

4  LD BC,1000H
```

This is called a **source listing** and obviously if you knew what the different statements meant you would be able to guess what the program was supposed to do.

Once the **source code** has been written we can tell the **assembler** to **Assemble** the code into machine language instructions, and within a few seconds it would produce a listing similar to the following:-

LISTING 5

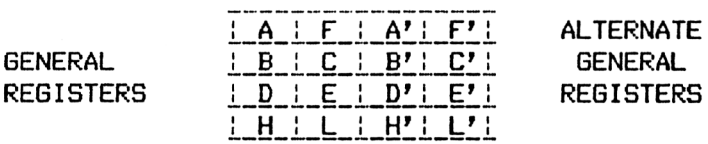
```
1                ORG   0ACFEH
2  ACFE  211010        LD HL,1010H
3  AD01  CD75BB        CALL 0BB75H
4  AD04  01E803        LD BC,1000H
```

Please note that the assembler used to write the routines in this book is Kuma's ZEN. Assemblers work in different ways, so if you are using another product, your listings may vary slightly from the examples shown here.

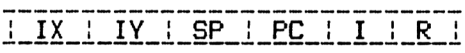
We use an assembler to write assembly language programs written in Z.80 mnemonics, which are easy to understand logical instructions. The assembler turns this program called a **source program** into machine code instructions that the computer can understand. The resultant code is called the **object code**.

```
SOURCE CODE  ===> ASSEMBLER ===> OBJECT CODE
```

The Z80 processor contains two sets of internal, general registers, and six special purpose registers. Take a look at the following:-



Z80 REGISTERS



SPECIAL PURPOSE REGISTERS

A,B,C,D,E,F,H,and L are the normal general registers and the registers designated ' are the **alternate** register set, which can only be accessed by the two instructions **EX AF,AF'** AND **EXX** - these two instructions exchange the contents of the main set with that of the alternate set. Only one set of registers can be used at one time. Following the two sets of **8Bit** registers are four **16Bit** registers : **IX,IY,SP,PC.** Registers **I & R** are very seldom used in most normal programming applications.

The **A** register is also refered to as the **accumulator** because all of the arithmetic instructions, and most of the other instructions use the contents of the **A** register as an operand. In fact, this is where most of the transfers take place.

The **F** register is also called the **Flag** register. The **F** register sets, or re-sets, bits internally to indicate a **true or false** type of condition and is **never** used for computations.

The **Program Counter** or **PC** is a 16 bit register that points to the current memory location which hold the instruction to be executed. Another 16 bit register is the **SP** or **Stack Pointer**. This register keeps a check on the position of the **STACK** in RAM. Two 16 bit registers with very powerful programming possibilties are the **index registers**; **IX & IY**.

Each of the 8 Bit registers can be used separately or in set pairs [BC,DE,HL] and treated as 16 Bit registers.

Assemblers have their own set of rules, but they aren't difficult to learn:-

```
DB or DEFB  =====>     Define  Byte.......DB   #FA,'T','ONE'

DW or DEFW  =====>     Define Word.......DW #4007 or Label

DS or DEFS  =====>     Define Space......DS 245  reserve  245 bytes

DM or DEFM  =====>     Define Message....DM 'ANOTHER GAME ?'
```

All the above are known as **Pseudo operations** and are used by the assembler, **not the CPU**, to carry out predefined functions.

LABELS are used to reference one instruction to another. For example: **JP Z,AGAIN**. A label can be compared with a line number in Basic; e.g **IF A = 0 THEN GOTO 100**.

The semi-colon ; is used in the same way as the REM statement in Basic, and the assembler ignores all that follows. It is good programming practice to get into the habit of documenting your program. Believe me, when you look at your code after a few months you will find it hard to understand what you had in mind at the time you wrote the program.

The convention used by most assemblers is as follows:-

```
Label       Op Code       Operand       Remarks

START:      LD A,         (DE)          ;Put score in A reg.
```

A common CPU operation is the **compare** instruction - **CP in Z80 mnenomics. This works in a similar way to the Basic statement:-**

```
10 IF A = 10 OR A > 10 OR A < 10 THEN GOTO 100
```

It allows you to make decisions then act accordingly by branching out of one routine, or jumping into another part of your program. The result of a **compare** is checked by testing the state of the **bits** in the **F** register.

```
BIT  7  6  5  4  3  2  1  0
     S  Z  -  H  -  P/V N  C
```

```
C  =  CARRY FLAG : N = ADD/SUBTRACT FLAG [ BCD OPERATIONS ]
H  =  HALF CARRY FLAG [ BCD OPERATIONS ] : P/V = PARITY OVERFLOW
          Z  =  ZERO FLAG  : S  =  SIGN BIT
```

Bits **3 & 5** are not used. The **Half Carry and N flags** are used for **Binary Coded Decimal** operations, and we are not concerned with them at this point.

The **Carry Flag**, if set, denotes a **CARRY (C)**, and if reset denotes a **NO CARRY (NC)** condition. This flag is directly affected by an addition or subtraction. It should be understood that all **CP** operations compare the value contained in the **A register** with the next operand, which can be a **value in a register** or an **absolute value**:

```
CP C    ; compare value in A with value in C

CP #32    ; compare value in A with 32 hexidecimal
```

What is actually happening during a compare operation is the value of the compare operand is subtracted from the value contained in the A register.

```
LD A,#40      ; Load A reg with 40 Hex

CP L          ; VALUE OF A - VALUE OF L
```

You can see, from the above,that a **compare** operation is essentially an arithmetic operation on the A register, and, as such, the result will affect the **Carry Flag**.

The **Zero flag** is set [1] whenever the result of an arithmetic operation results in **zero**. If the **Carry & Zero flags** are used in tandem, any possibility can be tested. Consider the following Basic statement:-

```
10 LET A = VALUE
20 IF A = 10 THEN GOTO 40
30 IF A> 10 THEN GOTO 50
40 GOTO 40
```

Translating this into assembler:

```
     LD A,VALUE       ; put value in A reg.
     CP 10            ; compare value in A to 10.
     JR Z, EQUAL      ; if value in A = 10 then goto Equal.
     JR NC,GREATER    ; if carry flag  not set goto Greater.
LOOP:JR LOOP          ; value in A is not equal to, and is less than 10
```

The **No Carry** situation will arise if A = 10 or A > 10 and so it is always wise to compare the A register with a value **1 greater than the value you wish to test for.**

```
CP 10

JR NC,NEXT
```

If the carry is not set then A is definitely greater than 9 but could be equal to 10.

This is the reason we tested for **Zero** before testing the carry flag in the previous example.

The four situations can be summarised as follows:-

```
N  Value in A reg > or = to compared value
C  Value in A reg < compared value
Z  Value in A reg = compared value
NZ  Value in A reg not equal compared value
```

Also note that the value in the A register is not affected and is left unchanged by the compare.... the subtraction takes place in theory only. **The Sign Bit**: In 2's complement notation,if the 7th bit is a **1** then the number is **negative**, and if bit 7 is = 0 then the number is

positive. The **sign bit** reflects the state of this seventh bit. The other flags will be discussed as the situation arises, but the three already discussed are the most important.

ADDRESSING MODES

Any detailed review of a CPU will always mention its addressing modes. This is where the Z80 comes into its own: the wide variety of addressing modes available on this CPU makes life really easy for the programmer. Addressing modes will create no serious problem to you. You will soon become familiar with the most useful, and to help you, here are the most common ways of addressing the Z80.

Immediate Addressing:

In Basic a similar instruction would be : **LET A = 3**

LD A,03 or **LD HL,5007** (known as immediate extended addressing)

You are loading a register or a register pair with immediate data.

Register Addressing:

This is exactly what it says: One register is loaded from another.

LD A,C ; Load A from C

Indirect Register Addressing:

In this mode of addressing, the location of the operand is

held in one of the register pairs; **BC,DE or HL.** A translation in Basic would be:

10 LET BC = 14390
20 LET A = PEEK(BC)

In assembler:

```
LD A,(BC)      ; load A register with the value in
                 the RAM/ROM location
               ; pointed to by the BC register pair.
::::::::::
LD HL,14390    ; make HL point to address 14390
LD A,(HL)      ; put value in A register.
LD DE,56789    ; point DE registers to location 56789
LD (DE),A      ; load memory location pointed to by DE
                 registers with
               ; value in A register.
```

Indexed Addressing:

This is a really powerful addressing mode. It allows you to retrieve, or store data, from tables set up in memory. We can make **IX or IY** registers point to an address then add an **offset** within the range of **-128 to +127.**

If the IX register points to memory address 3C00 Hex we can **LD A,(IX+15)** which would load the A register with the contents of memory location **C0F hex.** LD A,(IX+00) would load the A register from memory location **3C00 hex**.

Implied Addressing:

This mode means that the register is not named in the mnemonic, but is implied by it.

```
ADD E   ;The contents of the E register are added to
the contents of  the ;A register.
SCF     ; Set Carry Flag.
```

```
Protocol:

LD A,(HL) ........  LD A FROM LOCATION POINTED TO BY HL.
Data flow....    A <====== HL <========== Memory Location.
Data always flows from RIGHT to LEFT  LD A,36
```

DJNZ

The DJNZ instruction works in tandem with the B register and is used in a similar manner to a FOR NEXT LOOP in Basic.

```
          LD B,100

LOOP:     DEC HL

          LD (HL),A

          DJNZ LOOP ............. FOR I = 1 TO 100:

          PRINT I:NEXT
```

Chapter Six

A Choice Remark

Chapter Two saw us dealing with binary numbers and powers of two. Understanding the binary number system can help us with our programming in all sorts of ways, and it certainly makes life a lot easier when dealing with graphics.

Each location or print position on the Text Screen is made up of 8 pixel by 8 pixel squares. **Mode 1** has 40 print positions on each of the 25 lines which makes a total of (40*25*8) 1000 pixels used by the Text screen. When a character is printed on the screen it covers one of these 8 by 8 pixel positions, the parts that we see are the lit pixels and the blank parts of the character are the unlit pixels.

All characters, **Ascii** or graphics, follow this same format of 8 x 8 pixel squares. The character can be a solitary lit pixel (Full stop) or 64 lit pixels which should as a solid block on the screen. This method of displaying graphics relates very well to the binary number system. If we treat each lit pixel as a **1** (on) and each unlit pixel as a **0** (off), we can convert each design into a set of data for sending to the computer with the **Symbol** command.

When converting Data into lit and unlit pixels for displaying on the screen the computer uses the same

Positional Notation as used for numbering the bits of a byte.

```
                  2⁷  2⁶  2⁵  2⁴  2³  2²  2¹  2⁰
Byte Format:  ---------------------------------
              ! 1 ! 1 ! 0 ! 0 ! 1 ! 0 ! 1 ! 1 !
              ---------------------------------
```

CHARACTER MATRIX OF X

```
      2⁷  2⁶  2⁵  2⁴  2³  2²  2¹  2¹
     ---------------------------------
  0  !***!***!   !   !   !***!***!   !
     ---------------------------------
  1  !   !***!   !   !   !***!   !   !
     ---------------------------------
  2  !   !   !***!   !***!   !   !   !
     ---------------------------------
  3  !   !   !   !***!   !   !   !   !
     ---------------------------------
  4  !   !   !***!   !***!   !   !   !
     ---------------------------------
  5  !   !***!   !   !   !***!   !   !
     ---------------------------------
  6  !***!***!   !   !   !***!***!   !
     ---------------------------------
  7  !   !   !   !   !   !   !   !   !
     ---------------------------------
```

To convert a character design into data we simply add all the **positional** values of the **1s** (lit pixels) in each horizontal line, and the result is the number used to reproduce that line.

```
      2⁷  2⁶  2⁵  2⁴  2³  2²  2¹  2¹
     ---------------------------------
0    !***!***!   !   !   !***!***!   ! = 128 +64 + 4 + 2 = 198
     ---------------------------------
1    !   !***!   !   !   !***!   !   ! = 64 + 4  = 68
     ---------------------------------
2    !   !   !***!   !***!   !   !   ! = 32 + 8  = 40
     ---------------------------------
```

Part of the X design

Symbol 128, 198,68,40 ...,... would create the top three lines of the X and assign the value to character 128.

Larger patterns can be created by using two or more matrices butted together to give 16 * 16 pixel characters - in fact you can use all 255 characters and create a unique and exclusive design.

```
10 CLS

20 DATA 175,215,175,213,172,212,172,212

30 DATA 252,0,80,248,112,32,0,0

40 DATA 192,248,254,254,30,28,60,56

50 DATA 56,112,112,112,126,49,65,126

60 CHR=128

70 FOR I = 0 TO 3

80 READ A,B,C,D,E,F,G,H

90 SYMBOL CHR,A,B,C,D,E,F,G,H

100 NEXT

120 X$ =CHR$(128)+CHR$(130)

130 Y$=CHR$(129)+CHR$(131)

140 LOCATE 12,12

150 PRINT X$;CHR$(8):CHR$(8);CHR$(10);Y$

160 GOTO 160
```

The above code uses two control characters **CHR$(8) & CHR$(10)** to backspace the cursor and perform a line

feed so that the Y$ will be printed immediately under the X$. This will give the effect of printing a single character 16 pixels across by 16 pixels deep.

Movement can be achieved, in it's simplest form, by first printing the character, then over printing it with a space, the print position is then incremented and the same procedure repeated. Some kind of a delay loop must be used so that movement appears fluid.

```
Print character      Print character      Print character
Delay                Delay                Delay
Print space--------> Print space -------> Print space
```

```
10 CLS: Y=1

20 FOR X = 1 TO 39

30 LOCATE X,Y

40 PRINT "A";

50 FOR J = 1 TO 80: NEXT

60 PRINT CHR$(8);" "; ' BACKSPACE PRINT POSITION

70 NEXT
```

Animation can be simulated by designing two separate patterns and printing them alternately to the screen. Again, a delay loop will have to be used to give a smooth and realistic animation. If the above method of movement is also employed, the appearance of a man walking or a bird flying etc., can easily be achieved.

The way the Amstrad's screen is mapped in memory makes it virtually impossible to read directly from the screen using the Peek statement. This is an unfortunate

situation as it is sometimes desirable to know what character is at the cursor position. The following program will provide you with the facility to Peek the Amstrad screen at the current cursor position and returns the value in integer variable **PK%**. **PK%** must be defined before calling the routine otherwise unpredictable results will occur. The program also introduces yet another way of storing machine code routines.

```
1 GOTO 10:REM**************

10 CLS:DEFINT A-Z:PK=0:AD=&017B

20 DATA &DD,&6E,&00,&DD,&66,&01,&E5

30 DATA &CD,&60,&BB,&E1,&77,&C9

40 FOR I = AD TO AD+12

50 READ DTA:POKE I,DTA

60 NEXT

70 LOCATE 4,12:PRINT "A"

80 LOCATE 4,12:CALL AD,@PK

90 LOCATE 1,1:"PRINT PEEK VALUE IS ";PK

100 GOTO 100
```

The program gets whatever character is at the current cursor location and puts it into variable PK. by examining the contents of PK you have effectively Peeked the screen.

This method uses a **REM** statement to store the machine code driver. whenever you use this method you must set up **Line 1 exactly as shown in the above program**

except that you must put as many * in the **REM** statement as there are bytes in your machine code program. If you set up the line exactly your machine code will always start at **&017B1, and can be called from this location.**

One final point. Always make sure you save your program before attempting to **RUN** it, as it will be impossible to edit it afterwards. If you try to list the program after running, you will get a **SYNTAX ERROR**. This can be overcome by typing **LIST 10 -** which will then list all the lines from line 10.

This is a very versatile method of storing short machine code subroutines, and providing you follow the instruction above,the program will be self contained and cause no problems.

THE AMSTRAD SCREEN MAP

Line 1 $C000_H$ ..... $C04F_H$
$C800_H$ ..... $C84F_H$
$D000_H$ ..... $D04F_H$
$D800_H$ ..... $D84F_H$
$E000_H$ ..... $E04F_H$
$E800_H$ ..... $E84F_H$
$F000_H$ ..... $F04F_H$
F800H ..... $F84F_H$

Line 2 $C050_H$ ..... $C09F_H$
$C850_H$ ..... $C89F_H$
D050H ..... $D09F_H$
$D850_H$ ..... $D89F_H$
$E050_H$ ..... $E09F_H$
$E850_H$ ..... $E89F_H$
$F050_H$ ..... $F09F_H$

Line 3 $C0A0_H$ ..... $C0EF_H$
: :
: :
$F8A0_H$ ..... $F8EF_H$

Line 4 $C0F0_H$ ..... $C1BF_H$
: :
: :
$F8F0_H$ ..... $F93F_H$

Line 5 $C140_H$ ..... $C18F_H$
: :
: :
$F940_H$ ..... $F98F_H$

: :

Line 25 $C780_H$ ..... $C7CF_H$
$CF80_H$ ..... $CFCF_H$
$D780_H$ ..... $D7CF_H$
$DF80_H$ ..... $DFCF_H$
$E780_H$ ..... $E7CF_H$
$EF80_H$ ..... $EFCF_H$
$F780_H$ ..... $F7CF_H$
$FF80_H$ ..... $FFCF_H$

==========

Chapter 7

SOUND ADVICE

A really exciting innovation of the Amstrad computer is the way it has been designed to allow the programmer easy access to modify the operating system. Entry to the system is via a series of **Jump Blocks** placed in high memory starting at $B900_H$

The **Jump Blocks** contain a series of vectors to the more useful routines contained in the **Firmware Rom**. The lower Rom, as we have already intimated, is responsible for controlling the hardware and related routines. By using the vectors so kindly supplied by the brains at Locomotive Software we can take advantage of the Rom routines instead of having to write thousands of lines of our own. Because the **Jump Blocks** are in Ram we can easily make slight alterations, or modify the routines in a more drastic manner, by re-vectoring the addresses within the **Jump Block**.

To take full advantage of these routines it is advisable to purchase a copy of Amstrad's "**The Complete CPC 464 Operating System**" (Soft 158). The book is expensive but it contains a wealth of information on the operating system, and is virtually the Amstrad programmer's **Bible**. There are over 189 different entries in the **Jump Blocks, and it is well beyond the scope of this book to deal with each individual call.**

To demonstrate the usefulness of the Jump Blocks let's write a small routine to print a message on the screen. Not very enterprising, I agree, but it will illustrate how easy it is to utilise the Rom routines.

```
        ORG  0A00H
SETCUR:EQU  0BB75H
PRINT: EQU  0BB5DH
;
START:  LD HL,0104H
        CALL SETCUR
        LD HL,MESS
AGAIN:  LD A,(HL)
        CP 0FFH
        JR Z,FINISH
        PUSH HL
        CALL PRINT
        INC HL
        JR AGAIN
FINISH: RET
MESS: DB "THIS IS A TEST PROGRAM FOR THE
AMSTRAD",0FFH
```

The program calls on two entries from the **Jump Block: $BB75_H$** and **$BB5D_H$**. At the very start of the program the two addresses are given labels with the actual addresses EQUated immediately on entering the routine.

SETCUR $BB75_H$

The routine called from this address moves the cursor to a new location as specified in the HL register pair. H must contain the required column and L the desired row.

PRINT $BB5D_H$

Prints the character contained in the A register at the current cursor position.

The program starts by loading the HL register pair with the desired cursor position (LOCATE 1,4). The HL registers are then pointed to the address of the message buffer (LD HL,MESS), and a loop is set up (AGAIN) to load the A register with the characters pointed to by HL. A test is carried out to determine if the contents of the Accumulator are equal to FF_H - this is the **flag byte**. We have used a similar method in Basic for detecting the end of Data. If the character is not FF_H the PRINT routine is invoked to display the message on the screen. Program flow then jumps back to the AGAIN loop to search for the next character. Once the FF_H marker is found the routine terminates by returning to the calling program.

LISTING 5

The assembled program.

```
 1                          ORG  0A000H
 2              SETCUR:     EQU  0BB75H
 3              PRINT:      EQU  0BB5DH
 4              ;
 5 A000 210401  START:      LD   HL,0104H
 6 A003 CD75BB              CALL SETCUR
 7 A006 2116A0              LD   HL,MESS
 8 A009 7E      AGAIN:      LD   A,(HL)
 9 A00A FEFF                CP   0FFH
10 A00C 2807                JR   Z,FINISH
11 A00E E5                  PUSH HL
12 A00F CD5DBB              CALL PRINT
13 A012 23                  INC  HL
14 A013 18F4                JR   AGAIN
15 A015 C9      FINISH:     RET
16 A016 54484953 MESS:      DB   "THIS IS A TEST PR
16 A01A 20495320
16 A01E 41205445
16 A022 53542050
16 A026 524F4752
16 A02A 414D
17 A02C FF                  DB   0FFH
18                          END
```

Once we have assembled the program we can install it in a number of ways:

a] By loading it into protected memory and CALLing it from the Basic program.

b] By saving the machine code in Basic Arrays and Pokeing it into protected memory, then calling it from within the Basic program.

c] By loading it into a string, and Calling the Address of the string.

d] If the code is re-locatable we can load it into a Basic Array as described in Chapter 5, and run it by calling the Address of the Array.

On this occasion we shall place the instructions into Data statements, and Poke them into memory, then Call the routine from Basic.

```
10 CLS

20 FOR I = &A000 TO &A02C

30 READ DTA

40 POKE I,DTA

50 NEXT

60 CALL &A000H

70 FOR I = 1 TO 500:NEXT

80 CLS:GOTO 60

90 DATA &21,&04,&01,&CD,&75,&BB,&21,&16,&A0

100 DATA &7E,&FE,&FF,&28,&07,&E5,&CD,&5D,&BB

110 DATA &23,&18,&F4,&C9,&54,&48,&49,&53,&20

120 DATA &41,&20,&54,&45,&53,&54,&20,&50,&52

130 DATA &4F,&47,&52,&41,&4D,&FF
```

Now that we have sampled the way a **Jump Block** can be used to our advantage, let's move on to do something more useful, and at the same time, design a routine to give our programs more appeal.

SOUND

It is not our intention to dwell on the inbuilt sound routines as they are adequately covered in the **Concise Basic Manual** published by Amsoft. We are going to design our own sound routines and create **3 new Basic commands** to allow us to take full advantage of them.

It would be nice to have a less complicated way to produce sounds on the computer. The program we are about to design will give us this facility. When we have finished the overlay, we will be able to use 3 new commands:

|SOFF, |PLAY, and |SND.

To write this program we must first understand how the system utilises the sound chip, and how the AY8912 Programmable Sound Generator operates. Obviously, without knowing this information, it would be impossible to write an effective routine to interface with the chip.

AY8912 Programmable Sound Generator [PSG]

The PSG is a **Large Scale Intergrated 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 Amstrad, 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 frquencies 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 for each 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_n],** and their functions within the PSG are listed as follows.

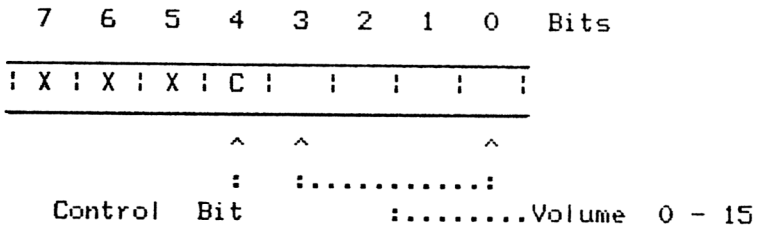
THE PSG REGISTERS

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 [1]. Control of the channels is by 5 bits only (0 to 4) and the remaining 3 bits are not considered by the PSG.



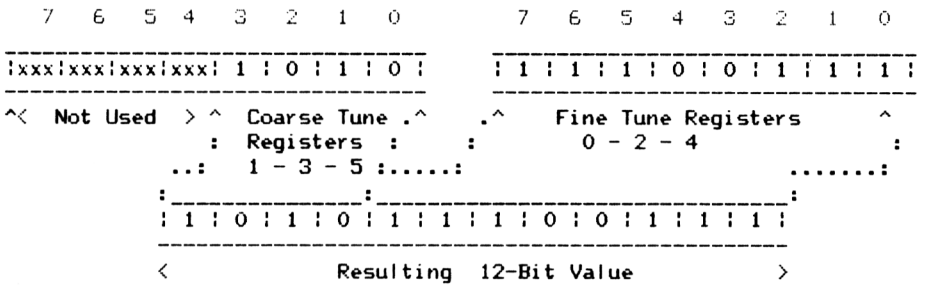
If **BIT 4** is set [1], then control is passed to the **Envelope Generator** which gives a variable level of sound combined with the differing waveforms.

When **BIT 4** is reset [0], control is by the value passed in registers 8,9,& 10. All this means is: if you pass a value between 0 to 15 then the volume level is that value,with 0 the lowest,and 15 the highest volume. When the value in these registers is 16 the volume varies under control of **Register 13** - 16 sets BIT 4.

REGISTERS 0,1,2,3,4,5

These registers control the pitch of the note produced. The lower channel of each pair uses BITS 0 - 7 and governs the **Fine Tune** (just using the lower channel produces high pitched notes). The higher channel of each

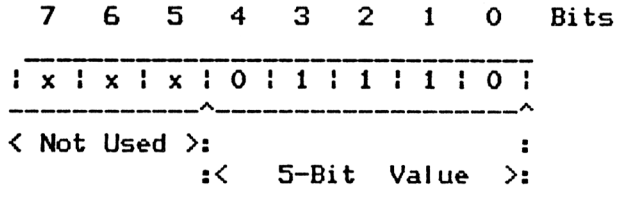
pair uses 4 BITS 0 to 3, and controls the **Coarse Tune** (low pitched sounds). When the values of the two registers are combined, the result is a 12-bit,value and the note produced as listed in your Basic Manual for that value.



The combined value is equivalent to the note value used in Basic

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 to 4), and the higher the value in the register the lower the resulting frequency of the white noise. The range of values that can be input to this register is 0 to 31_D.



REGISTER 7

As far as we are concerned, this is the most important register to master, as this register mixes noise and tone for registers **8,9,& 10**.

The register is also bit specific and if you study **Table 7.1** you should easily understand how different values affect the final output on channels A,B, and C.

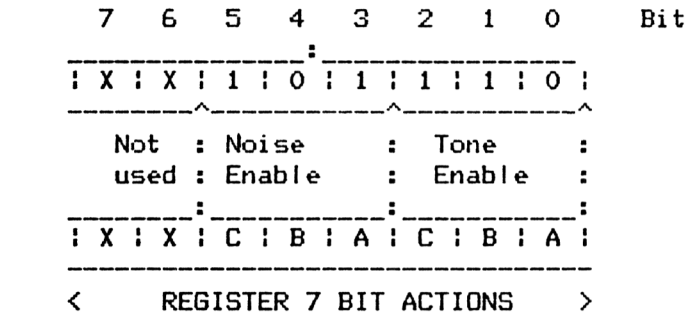


TABLE 7.1

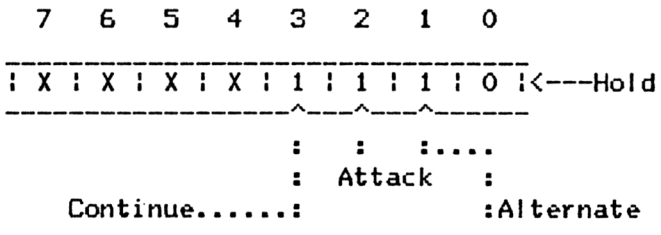
Result	Channels	Bits 5 4 3	2 1 0	Result	Channels
Noise on	A B C	0 0 0	0 0 0	Tone On	A B C
Noise on	- B C	0 0 1	0 0 1	Tone On	- B C
Noise on	A - C	0 1 0	0 1 0	Tone On	A - C
Noise on	- - C	0 1 1	0 1 1	Tone On	- - C
Noise on	A B -	1 0 0	1 0 0	Tone On	A B -
Noise on	- B -	1 0 1	1 0 1	Tone On	- B -
Noise 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 on Channel B,and the sound on Channels A & C,you need to look in the tables and combine the necessary values. E.g $101010_B = 42_D$

Bits 6 and 7 control the Input/Output register. This register is used by the computer for keyboard scanning and does not concern us. The range of values that can be passed is in the range 0 to 255_D, and remember that the bit must be re-set [0] to enable the channel.

REGISTER 13

This register uses 4 bits (0 to 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.



HOLD

When Bit 0 is 1 the envelope is limited to one cycle.

ALTERNATE

If Bit 1 is set then the envelope counter reverses direction after each cycle.

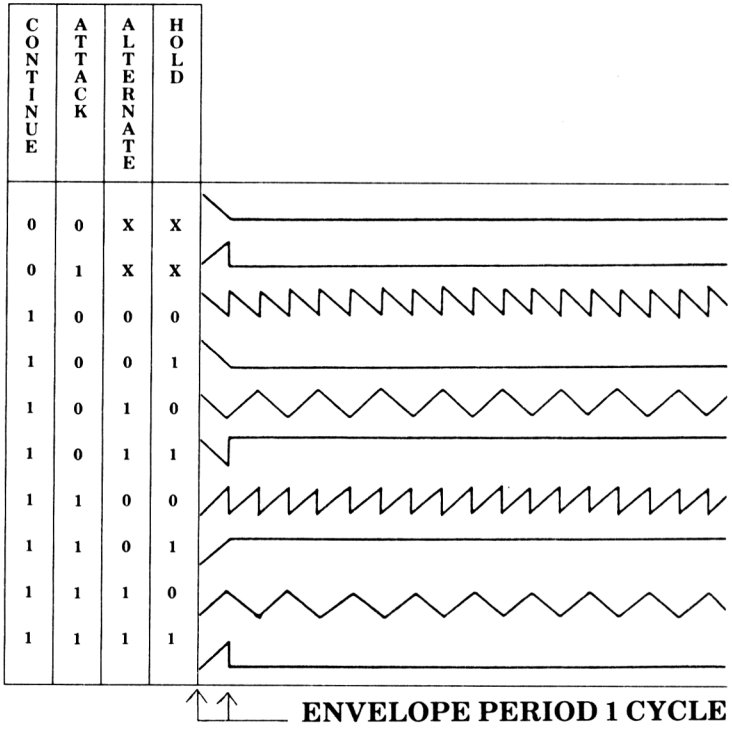
ATTACK

When set to 1 = Attack else = DECAY.

CONTINUE

When Bit 3 is set the **cycle** pattern will be defined by the **Hold** bit.

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 (2 bytes) and can have a value in the range 0 - 65535_D. Register 12 controls the **Fine Tune** (high pitched notes), and Register 11 controls the enveloped period of the **Coarse Tune** (low pitches).

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 the two registers can produce some interesting sound effects from the bouncing of a ping-pong ball to the sound of a racing car.

ADDING THREE NEW SOUND COMMANDS TO BASIC

After the program in **Assembly Listing One** has been installed in the computer, Basic will be able to respond to 3 new commands: **!PLAY, !SND, and !SOFF.**

FORMAT FOR :PLAY

```
!PLAY,<Channel>,<Octave>,<Note>,<Volume>

Channel = 1 to 3

Octave  = 1 to 8

Note    = 1 to 12

Volume  = 1 to 15

MIDDLE C = OCTAVE 4
```

NOTES:

0 = REST	4 = F	8 = C_{sharp}
1 = C	5 = G	9 = D_{sharp}
2 = D	6 = A	10 = F_{sharp}
3 = E	7 = B	11 = G_{sharp}
		12 = A_{sharp}

FORMAT FOR :SND

!SND,<Register Number>,<Value>

Registers = 0 to 13
Value = 0 to 255

!SOFF Turn all sound channels off

```
 1                ;**********************
 2                ;ASSEMBLY LISTING ONE
 3                ;**********************
 4                ;
 5                ;
 6                          ORG  42000
 7                          LOAD 42000
 8                SEND:     EQU  0BD34H          ;JP TABLE CALL
 9                OFF:      EQU  0BCA7H          ;JP TABLE CALL
10 A410 011AA4             LD   BC,COMTAB       ;COMMAND TABLE
11 A413 21BEA4             LD   HL,BUF          ;REQUIRED BY BASIC
12 A416 CDD1BC             CALL 0BCD1H          ;LOG ON NEW COM TAB
13 A419 C9                 RET                  ; LOGGED ON NOW
14                ;             TO BASIC
15 A41A 25A4      COMTAB:  DEFW NMETAB
16 A41C C331A4             JP   SND             ; SND COMMAND
17 A41F C347A4             JP   PLAY            ;PLAY COMMAND
18 A422 C3C3A4             JP   SOFF            ;SOUND OFF CALL
19                ;****************************
20                ;Define table of new commands
21                ;so that Basic knows what to
22                ;expect.
23                ;****************************
24                ;
25 A425 534E      NMETAB:  DEFB "SN"
26 A427 C4                 DB   "D"+80H
27 A428 504C41             DB   "PLA"
28 A42B D9                 DB   "Y"+80H
29 A42C 534F46             DB   "SOF"
30 A42F C6                 DB   "F"+80H
31 A430 00                 DB   00H
32                ;
33                ;
34                ;*********************************
35                ; SND ROUTINE STARTS HERE
36                ; DIRECT ACCESS THROUGH CPU TO
37                ;SOUND CHIP REGISTERS
38                ;*********************************
39                ;
40                ;
41 A431 DD7E02    SND:     LD   A,(IX+02H)      ;GET REGISTER NO
42 A434 F5                 PUSH AF              ;REGISTER ON STACK
43 A435 FE07               CP   07H             ;COMPARE CARRIES OVER
44 A437 DD7E00             LD   A,(IX+00H)      ;VALUE
45 A43A 2002               JR   NZ,NAND         ;FROM ABOVE COMPARE
46 A43C E63F               AND  3FH             ;MAKE SURE 10111111
47 A43E 5F        NAND:    LD   E,A             ;VALUE NOW IN E REG
48 A43F F1                 POP  AF              ;REGISTER OFF STACK
49 A440 FE0E               CP   0EH             ;13 IS MAXIMUM VALUE
50                ;          COMPARE WITH MAX+1
51 A442 304B               JR   NC,REST         ;>= 14 IS TOO BIG
52 A444 C39DA4             JP   OUT2            ;SO GO BACK
53                ;
54                ;
55                ;*********************************
56                ;START OF PLAY COMMAND ROUTINE
57                ;FORMATTED AS CHAN/OCT/NOTE/VOL
58                ;*********************************
59                ;
60                ;
```

```
 61 A447 DD7E06   PLAY:      LD   A,(IX+06H)       ;GET CHAN AND
 62 A44A F5                  PUSH AF               ;SAVE IT
 63 A44B DD7E04              LD   A,(IX+04H)       ;GET OCT
 64 A44E F5                  PUSH AF               ;AND SAVE IT
 65 A44F DD7E02              LD   A,(IX+02H)       ;GET NOTE
 66 A452 B7                  OR   A                ;CHECK NOT 0
 67 A453 283A                JR   Z,REST           ;IF IT IS JP
 68 A455 CB27                SLA  A                ;MULTIPLY BY 2
 69 A457 5F                  LD   E,A              ;NOW IN E REG
 70 A458 1600                LD   D,00             ;DE = NOTE
 71 A45A 21A2A4              LD   HL,TABLE         ;POINT HL
 72 A45D 19                  ADD  HL,DE            ;HL NOW = NOTE
 73                                 ;IN TABLE
 74 A45E 5E                  LD   E,(HL)           ;GET LSB
 75 A45F 23                  INC  HL               ;ALIGN TO GET
 76 A460 56                  LD   D,(HL)           ;MSB IN D
 77              ; ***** DE NOW = CORRECT NOT VAL
 78 A461 C1                  POP  BC               ;GET OCTAVE OFF
 79                                 ;STACK
 80 A462 0E00                LD   C,00
 81 A464 05                  DEC  B
 82 A465 2806                JR   Z,SKIP           ;IF B = 0 SKIP
 83 A467 CB3A     ROTA:      SRL  D                ;ELSE ALIGN NOTE
 84                                 ;FOR CORRECT
 85                                 ;OCTAVE
 86 A469 CB1B                RR   E
 87 A46B 10FA                DJNZ ROTA             ;DO IT AGAIN
 88                                    ;UNTIL B = 0
 89 A46D F1       SKIP:      POP  AF               ;GET CHAN IN A
 90 A46E F5                  PUSH AF               ;SAVE IT AGAIN
 91 A46F CB27                SLA  A                ; ALIGN TO REG
 92                          ; IF CHAN = 1 THEN
 93                          ; *2 = REGISTER 2
 94                          ; FOR COARSE TUNE VAL
 95 A471 3D                  DEC  A                ;LESS 1 NOW CORRECT
 96 A472 CD96A4              CALL OUT1             ;GO SEND IT
 97 A475 DD7E00              LD   A,(IX+00H)       ;GET VOL
 98 A478 FE10                CP   10H              ;MAKE SURE NOT MORE
 99 A47A 3802                JR   C,OK             ;THAN 16
100 A47C 3E10                LD   A,10H            ;IF IT IS MAKE IT 16
101 A47E 5F       OK:        LD   E,A              ;PUT IT IN E REG
102 A47F D5                  PUSH DE               ;SAVE IT ON STACK
103 A480 1E38                LD   E,38H            ;MAKE SURE REG 7 HAS
104 A482 3E07                LD   A,7              ; TONE CHANNELS ON
105 A484 CD9DA4              CALL OUT2             ;GO SEND IT
106 A487 D1                  POP  DE               ;GET VOL BACK
107 A488 F1                  POP  AF               ;GET CHANNEL BACK
108 A489 C607     RETRST:    ADD  A,7              ;ALIGN TO CORRECT
109                            ;REGISTER 8,9 or 10
110 A48B CD9DA4              CALL OUT2             ;GO SEND IT
111 A48E C9                  RET                   ;BACK TO BASIC!
112 A48F F1       REST:      POP  AF
113 A490 F1                  POP  AF
114 A491 1E00                LD   E,00
115 A493 C389A4              JP   RETRST
116 A496 6F       OUT1:      LD   L,A
117 A497 4A                  LD   C,D
118 A498 CD34BD              CALL SEND
119 A49B 2D                  DEC  L
120 A49C 7D                  LD   A,L
```

```
121 A49D 4B       OUT2:       LD   C,E
122 A49E CD34BD                CALL SEND
123 A4A1 C9                    RET
124 A4A2 0000     TABLE:       DEFW 0000
125               ;********** Table of notes
126 A4A4 EE0E                  DEFW 0EEEH
127 A4A6 4D0D                  DEFW 0D4DH
128 A4A8 DA0B                  DEFW 0BDAH
129 A4AA 2F0B                  DEFW 0B2FH
130 A4AC F709                  DEFW 09F7H
131 A4AE E108                  DEFW 08E1H
132 A4B0 E907                  DEFW 07E9H
133 A4B2 180E                  DEFW 0E18H
134 A4B4 8E0C                  DEFW 0C8EH
135 A4B6 8F0A                  DEFW 0A8FH
136 A4B8 6809                  DEFW 0968H
137 A4BA 6108                  DEFW 0861H
138 A4BC 0000                  DEFW 0000H
139               ;
140               ;
141               BUF:         DEFS 4
142               ;
143               ;
144 A4C2 00                    NOP
145 A4C3 CDA7BC   SOFF:        CALL OFF
146 A4C6 C9                    RET
147 A4C7 00                    NOP
148                            END
```

If we use the code as listed we simply load it into protected memory, CALL 42000_D, and then use the new commands from within our Basic program. There is, however, a disadvantage in using this method: every time we wanted to use the commands with a Basic program, we would have to load the routine into protected memory as a separate file, which is not a very satisfactory way of utilising the program.

The most useful method of using the program is to create a Basic loader by turning all the machine code into Data statements and Pokeing them into memory. If we use this method, any future programs can take advantage of the sound utility by merging with the Basic loader.

To create a set of Data we simply look at the **Object code** in the third column, and place **each pair** of Hex numbers into a Data statement.

```
10  DATA  &01,&1A,&A4,&21,&BE,&A4
```

The program makes three calls to the system via the Jump Block,**$BD34_H$**,**$BCA7_H$**,**and $BCD1_H$**. The most interesting of these calls is **$BCD1_H$**. Basic provides for the addition of new Basic commands in the form of: **|Command,<param>,<param<, ...** A new command must always be prefixed with a "|" which is the elongated version of the colon situated above the @ on the computer keyboard. You can test this by typing **PLAY <ENTER>**, and you will get a **Syntax Error**, error message. Now type **|PLAY <ENTER>**, and the computer will respond with an **Unkown Command** error.

To make sure that the computer does not reject our new commands the resident Rom must be informed that we intend to use the new keywords, and we do this by **Logging On** the new words via **$BCD1_H$**.

To Log On a new command word the first part of the program must follow the following format:

```
LD BC, Address of our JUMP TABLE

LD HL, Address of a 4 Byte Buffer  (required by Rom)

CALL 0BCD1H     ; LOG ON NEW COMMANDS
```

Our **Jump Table** must be set up as follows:

```
TABLE: DEFW Address of New Basic Commands

JP BASIC ROUTINE 1

JP BASIC ROUTINE 2

JP BASIC .......ETC
```

```
NEWWORDS:DEFM "PLA"
DEFB "Y"+80H
DEFM "SN"
DEFB "D"+80H
DEFB 00      ; Marks end of table.
```

Adding 80_H to the last character in each command word sets the high bit (7) of the character i.e Ascii **Y** + 80_H = $D9_H$. This convention is required so that the resident Rom can recognise the end of each new keyword.

Basic extensions can also pass parameters. They are passed to the handling routine by the **IX** register in the form:

```
(IX + 00)  points to last parameter.
(IX+ nn)   points to the first param.

!PLAY, Param1,Param2,Param3,Param4
          ^       ^       ^       ^
          :       :       :       :
(IX+03)...:       :       :       :...> (IX+00)
                  :       (IX+01)
                  (IX+02)
```

That's all there is to it! You can add as many new commands as you wish within the capabilities of memory.

Note: The first line of the Basic program must have a **Memory** statement that sets the top of Basic **one byte below** the routine start address. The **Sound routine**

has used $A410_H$ (42000_D) for the start address and this allows enough room to move the **Matrix Table from Rom to Ram with the Symbol After** command.

LISTING 6

Listing 6 is the Basic Driver program and this can be used with any program you wish. Once the Driver program has been run you can use the new commands in the **Direct Mode** to experiment with the sound registers.

```
10 CLS
20 FOR I = &A410 TO &A4C7
30 READ DTA
40 POKE I,DTA
50 NEXT
60 GOTO 290
70 DATA &01,&1A,&A4,&21,&BE,&A4,&CD,&D1,&BC
80 DATA &C9,&25,&A4,&C3,&31,&A4,&C3,&47,&A4
90 DATA &C3,&C3,&A4,&53,&4E,&C4,&50,&4C,&41
100 DATA &D9,&53,&4F,&46,&C6,&00,&DD,&7E,&02
110 DATA &F5,&FE,&07,&DD,&7E,&00,&20,&02,&E6
120 DATA &3F,&5F,&F1,&FE,&0E,&30,&4B,&C3,&9D,&A4
130 DATA &DD,&7E,&06,&F5,&DD,&7E,&04,&F5,&DD
140 DATA &7E,&02,&B7,&28,&3A,&CB,&27,&5F,&16
150 DATA &00,&21,&A2,&A4,&19,&5E,&23,&56,&C1
160 DATA &0E,&00,&05,&28,&06,&CB,&3A,&CB,&1B
170 DATA &10,&FA,&F1,&F5,&CB,&27,&3D,&CD,&96
180 DATA &A4,&DD,&7E,&00,&FE,&10,&38,&02,&3E
190 DATA &10,&5F,&D5,&1E,&38,&3E,&07,&CD,&9D
200 DATA &A4,&D1,&F1,&C6,&07,&CD,&9D,&A4,&C9
210 DATA &F1,&F1,&1E,&00,&C3,&89,&A4,&6F,&4A
220 DATA &CD,&34,&BD,&2D,&7D,&4B,&CD,&34,&BD
230 DATA &C9,&00,00,&EE,&0E,&4D,&0D,&DA,&0B
240 DATA &2F,&0B,&F7,&09,&E1,&08,&E9,&07,&18
250 DATA &0E,&8E,&0C,&8F,&0A,&68,&09,&61,&08
260 DATA &00,&00,&00,&00,&00,&00,&00,&CD,&A7
270 DATA &BC,&C9,&00
280 '*********************************
290 'REST OF PROGRAM CAN START HERE
```

DEMONSTRATION PROGRAMS

Now that you have your three new commands try typing in the following short programs, and you will soon see how versatile they are, and how they will enhance your future programs.

Demonstration One

```
10   FOR I = 48 TO 192
20   !SND,7,254
30   !SND,8,20
40   !SND,0,I
50   NEXT
60   !SND,6,0
70   !SND,7,7
80   !SND,8,30
90   !SND,9,30
100  !SND,10,30
110  !SND,12,56
120  !SND,13,0
130  FOR I = 1 TO 1000:NEXT
140  !SOFF
150  GOTO 120
```

Demonstration Two

```
10   INPUT "PRESS ANY KEY TO DETONATE"
20   !SND,6,0
30   !SND,7,7
40   !SND,8,16
50   !SND,9,16
60   !SND,12,56
80   !SND,13,0
90   GOTO 10
```

Demonstration Three

```
10   INPUT "PRESS ANY KEY TO FIRE"
20   !SND,6,15
30   !SND,7,7
40   !SND,8,16
50   !SND,9,16
60   !SND,10,16
70   !SND,12,16
80   !SND,13,0
90   GOTO 10
```

Chapter 8

AMSTRAD SPRITES

THE AMSTRAD DOES NOT HAVE SPRITES ! By the time you have finished this chapter this statement will no longer be true. The following program will allow us to **create up to 8 pseudo-sprites,and Peek the Amstrad screen.**

A sprite must be able to move around the screen in all directions without destroying the background design - moving over the background rather than on it. Also, as it is almost impossible to Peek the Amstrad screen, we must include in our Basic extension a routine that will overcome this problem so that sprite collisions can be detected.

Assembly Listing 2 is included so that you can see how the program is designed. Try to follow it through, it will give you some good ideas for new commands of your own.

The method of using the program is as before: Type in **Listing 7,** save it to tape. Now type in **Listing 8**,save it, then run the program. The demonstration module will show you what is possible with these routines - it is not meant to be an elite piece of programming, and I am sure

you will be able to improve it,which is the very reason it is included.

LISTING 7

```
2 'Sprite_loader.Bas
5 SYMBOL AFTER 32
10 MEMORY &9C39
20 FOR ADD = &9C40 TO &9E19
30 READ VL:POKE ADD,VL
40 NEXT
50 CALL &9C40
60 RUN"
100 DATA &01,&49,&9C,&21,&AE,&9D,&CD,&D1,&BC
110 DATA &54,&9C,&C3,&63,&9C,&C3,&5E,&9D,&C3,&8E,&9D
120 DATA &50,&55,&D4,&43,&52,&53,&50,&D2,&53,&43,&52
130 DATA &CE,&00,&00,&00,&DD,&7E,&02,&32,&B2,&9D,&CD,&4D,&9D
140 DATA &FD,&7E,&00,&FE,&FF,&C8,&DD,&7E,&00,&FE,&00
150 DATA &CA,&F4,&9D,&FE,&09,&D0,&FD,&77,&02,&CD,&78,&BB
160 DATA &22,&AB,&9D,&FD,&6E,&00,&FD,&66,&01,&E5,&CD,&75,&BB
170 DATA &FD,&7E,&04,&FE,&00,&20,&18,&3E,&FE,&FD,&77,&04
180 DATA &CD,&9F,&BB,&CD,&93,&BB,&32,&AD,&9D,&FD,&7E,&03
190 DATA &CD,&90,&BB,&E1,&C3,&1D,&9D,&FD,&7E,&05,&CD,&5D,&BB
200 DATA &3E,&FE,&CD,&9F,&BB,&CD,&93,&BB,&32,&AD,&9D
210 DATA &FD,&7E,&03,&CD,&90,&BB,&E1,&FD,&7E,&02,&FE,&01
220 DATA &28,&1D,&FE,&02,&28,&25,&FE,&03,&28,&1B,&FE,&04
230 DATA &28,&25,&FE,&05,&28,&10,&FE,&06,&28,&21,&FE,&07
240 DATA &28,&0E,&FE,&08,&28,&11,&C9,&25,&18,&17,&24
250 DATA &18,&14,&2D,&18,&11,&2C,&18,&0E,&2D,&25,&18,&0A
260 DATA &2C,&25,&18,&06,&2D,&24,&18,&02,&2C,&24,&7C,&FE,&01
270 DATA &38,&22,&FE,&29,&30,&1E,&7D,&FE,&01,&38,&19
280 DATA &FE,&1A,&30,&15,&FD,&75,&00,&FD,&74,&01
290 DATA &FD,&66,&01,&FD,&6E,&00,&CD,&75,&BB,&CD,&60,&BB
300 DATA &FD,&77,&05,&FD,&66,&01,&FD,&6E,&00,&CD,&75,&BB
310 DATA &FD,&7E,&06,&CD,&5D,&BB,&2A,&AB,&9D,&CD,&75,&BB
320 DATA &3E,&00,&CD,&9F,&BB,&3A,&AD,&9D,&CD,&90,&BB
330 DATA &C9,&3A,&B2,&9D,&3D,&07,&07,&07,&4F,&06,&00
340 DATA &FD,&21,&B3,&9D,&FD,&09,&C9,&DD,&7E,&08,&FE,&09
350 DATA &D0,&32,&B2,&9D,&CD,&4D,&9D,&DD,&7E,&02,&FD,&77
360 DATA &00,&DD,&7E,&04,&FD,&77,&01,&DD,&7E,&00
370 DATA &FD,&77,&03,&DD,&7E,&06,&FD,&77,&06,&3A,&B2,&9D
380 DATA &FD,&77,&07,&3E,&00,&FD,&77,&04,&C9,&DD,&7E,&00
390 DATA &6F,&DD,&7E,&02,&67,&7C,&FE,&01,&D8,&FE,&29,&D0
400 DATA &7D,&FE,&01,&D8,&CD,&75,&BB,&CD,&60,&BB
410 DATA &32,&F3,&9D,&C9,&00,&00,&00,&00,&00,&00,&00
420 DATA &00,00,00,00,00,00,00,00,00
```

```
430 DATA 00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00
440 DATA 00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00
450 DATA 00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00
460 DATA 00,00,00,00,00,00,00,00,00
470 DATA &CD,&78,&BB,&22,&AB,&9D,&FD,&6E,&00,&FD,&66,&01
480 DATA &CD,&75,&BB,&FD,&7E,&05,&CD,&5D,&BB,&3E,&FF
490 DATA &FD,&77,&00,&AF,&FD,&77,&01,&2A,&AB,&9D,&CD,&75
500 DATA &BB,&C9,&00
```

LISTING 8

```
1 'Sprite_Demo.Bas
2 '
3 'SET UP PARAMETERS ************
10 DEFINT A - Z: CLS:BORDER 9,9: ADD = &9DB3 ' Start Of SPRTBLE
11 INK 3,17
20 PK = &9DF3
30 FOR I =0 TO 56 STEP 8
40 POKE ADD+I,&FF
41 POKE ADD+I+4,0
42 NEXT
43 BUL =0:INVB =0
49 'DEFINE CHARACTERS **************
50 SYMBOL 249,&81,&42,&3C,&5A,&66,&3C,&42,&81
60 SYMBOL 250,&3C,&3C,&3C,&3C,&7E,&7E,&FF,&FF
70 SYMBOL 251,&10,&10,&0,&10,&10,&0,&10,&10
80 SYMBOL 252,&10,&10,&8,&8,&10,&10,&8,&8
85 PEN 1: FOR I = 3 TO 23
86 FOR J = 1 TO 3:X = INT(RND*39+1)
87 LOCATE X,I
88 PRINT "."
89 NEXT:NEXT
90 |CRSPR,1,249,10,2,3
100 |CRSPR,2,249,16,2,3
110 |CRSPR,3,249,23,2,3
120 |CRSPR,4,249,30,2,3
130 |CRSPR,5,250,20,24,2
140 FOR I = 1 TO 5:|PUT,I,1:NEXT ' Display cannon & invaders
149 'Main Program Loop **************
160 POKE PK,0:FL = INT(RND*9)
165 IF FL = 2 THEN GOSUB 3000:GOTO 190
170 IF FL = 5 THEN GOSUB 4000
190 IF INKEY(8) = 0 THEN 230
200 IF INKEY(1) = 0 THEN 250
210 IF BUL <> 0 THEN 1000
220 IF INKEY (47) = 0 THEN 2000
225 GOTO 1000
230 |PUT,5,1:GOTO 1000
250 |PUT,5,5:GOTO 1000
```

```
1000 IF INVB <> 0 THEN 1500
1010 B = INT(RND*4+1)
1020 B=B-1: B=ADD+(B*8)+1 ' Align to correct sprtble reference
1030 A = PEEK(B) ' Horizontal pos of inv
1040 A = A+1:!CRSPR,6,252,A,3,1:!PUT,6,7
1049 'Check human bullet
1050 INVB = 1:GOTO 1800
1500 CHK=PEEK(&9DDB):CHK1=PEEK(&9DDC):IF CHK >=25 THEN !PUT,6,
0:INVB =0: GOTO 1800
1505 CHK = CHK+1
1510 !SCRN,CHK1,CHK:IF PEEK(PK)=250 THEN 1520 ELSE IF PEEK(PK)
=251 GOTO 1515
1511 !PUT,6,7:GOTO 1800
1515 !PUT,6,0:!PUT,7,0:INVB=0:BUL =0:GOTO  160
1520 !PUT,6,0:!PUT,5,0
1530 INVB =0:BUL=0:!PUT,7,0
1540 LVE = LVE -1:IF LVE =0 THEN GOTO 5000
1550 FOR I= 1 TO 500:NEXT
1560 !CRSPR,5,250,24,20,2
1570 GOTO 160
1800 POSL =PEEK(&9DE3):POSH=PEEK(&9DE4):IF POSL = 1 THEN BUL
= 0:!PUT,7,0:GOTO 160
1805 POSL = POSL -1
1810 !SCRN,POSH,POSL
1820 IF PEEK(PK)=249 THEN 1830 ELSE IF PEEK(PK) = 252 THEN 1825
1821 !PUT,7,3:GOTO 160
1825 !PUT,7,0:!PUT,6,0:BUL=0:INVB=0:GOTO 160
1830 FOR I = 0 TO 40 STEP 8
1840   IF PEEK(ADD+I+1) <> PEEK(&9DE4) THEN NEXT:GOTO 160
1850 TMP = ADD+I+7
1870 INV =PEEK(TMP)
1880  !PUT,7,3
1900 !PUT,7,0
1910  !PUT,INV,0
1920 SCRE=SCRE+20
1930 IF SCRE = 80 THEN 5000
1940 BUL=0:GOTO 160
2000 BUL =1:X=0
2010 X=0
2020 X=PEEK(&9DD3)-1:Y=PEEK(&9DD4)
2030 !CRSPR,7,251,Y,X,2:!PUT,7,3
2040 GOTO 160
3000  FOR I= 1 TO 4
3010 !PUT,I,5
3020 NEXT
3030 RETURN
4000  FOR I= 4 TO 1 STEP -1
4010 !PUT,I,1
4020 NEXT
4030 RETURN
5000 STOP
```

Driver Program Notes:

!PUT

Will enable you to move a sprite,specified in the first parameter, in any direction,as nominated in the second parameter.

```
!PUT,<Sprite Number>,<Direction>
```

The sprite number can be 1 to 8, and the direction can be 1 to 8 with the sprite moving in the direction as shown in Table 8.1. **Note**: if a **zero** is placed in the <Direction> it will cause the sprite to be cleared from the screen and then place #FF [Sprite Not Created Marker] & 00 [First time routine entered Flag], in the Sprite Table. You must then re-create the sprite before you can display it on the screen. This is helpful when a collision with another object is detected and you wish to blank the sprite and leave the screen as before.

!CRSPR

To create a sprite you must use this command. The routine then stores the information into the correct entry point of the Sprite Table. The syntax for this command is:-

!CRSPR,<Sprite No>,<Patt No>,<X pos on screen>,<Y pos>,<Colour>

This command must be used to create your sprite before invoking the **!PUT** command. The sprite can now be displayed on the screen at the location specified in **!CRSPR** by issuing a **!PUT,<Sprite No>,1** command.

!SCRN

The **!SCRN** command peeks the screen and places the

result in location **$9DF3_H$**. Use the following syntax when invoking this command:-

```
!SCRN,<X position>,<Y position>
```

A typical basic line may look like this:-

10 !SCRN,3,5: LET PK = PEEK(&9DF3) Variables can be used with all commands e.g.

```
!SCRN,X,Y   or  !CRSPR,3,249,X,Y,3
```

This section of code can be detached and used as a **Screen Peek** utility with any other basic program.

SPRITE TABLE

The sprite table starts at location $9DB3_H$. The method of finding the correct displacement into the table is as follows:-

Sprite Table+(Sprite Number - 1 * 8):

If you wished to check in which direction sprite 3 is moving you would use the following: **&9DB3+(3-1*8+2)** to find the correct entry point.

The Sprite Table entries are as follows:

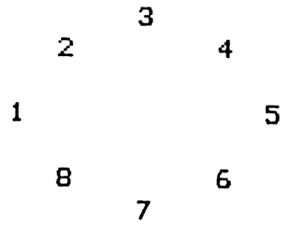
Byte 1	**Y Position**
Byte 2	**X Position**
Byte 3	**Direction**
Byte 4	**Ink**
Byte 5	**Flag**
Byte 6	**Basic Character**
Byte 7	**Sprite Pattern**
Byte 8	**Sprite Number.**

One point to note is,when two sprites collide erase the last sprite first by using the IPUT command followed by the sprite with which the collision occurred.

Listing 7 caters for most situations that could arise in your programming. By typing it in, and then studying how it works, you will gain a working insight on how to use these commands. The source listing can be used as a matrix with which you can create your own routines, and I would be interested to hear from any user who has found other commands which they have found useful.

Table 8.1

Movement of the sprite is in the direction as specified by the following arguments:



So that : PUT,1,5 will move sprite 1 to the left.

;ASSEMBLY LISTING 2

```
;               Program to add 3 New commands
;               to Amstrad CPC 464.
;
;
        ORG     40000
        ENT     $
        LD      BC,COMTAB
        LD      HL,BUF
        CALL    #BCD1           ;Log on New command table.
        ;
COMTAB:         DEFW    NMETAB
                JP      PUT
                JP      CRSPR           ;Additional new command
                JP      SCRN            ;Additional new command
                ;
NMETAB:         DEFB    "PU"
                DEFB    "T"+#80         ;#80 tells Amstrad Basic last
                DEFB    "CRSP"          ;letter in command name
                DEFB    "S"+#80         ;It sets BIT 7 HIGH.
                DEFB    "SCR"
                DEFB    "N"+#80
                DEFB    #00             ;End of table marker.
                DEFS    #02             ;Padding.
                ;
PUT:            LD      A,(IX+02)       ;Sprite Number
                LD      (SPRNO),A
                CALL    FINDISP         ;Find displacement into sprite table.
                                        ;Returns with IY pointing to first entry.
                LD      A,(IY+00)       ;L Position.
                CP      #FF             ;#FF signifies not yet created.
                RET     Z               ;Back to Basic if not created.
                LD      A,(IX+00)       ;PUT direction.
                CP      #00             ;00 means erase sprite   **new action
                JP      Z,BLANK
                CP      #09             ;Test if >8
                RET     NC              ;Jmp back to basic if yes.
                LD      (IY+02),A       ;Else save in sprite table.
                CALL    #BB78           ;Get current Basic csr pos.
                LD      (BASPOS),HL     ;Save it.
                LD      L,(IY+00)       ;Y POS
                LD      H,(IY+01)       ;X POS
                PUSH    HL              ;Save
                CALL    #BB75           ;Mve csr to our current position.
                LD      A,(IY+04)       ;Check flag byte in sprite table.
                CP      #00
                JR      NZ,SKIP         ;If not zero sprite has already been moved.
                LD      ,#FE            ;#FE is flag to say routine has
                LD      (IY+04),A       ;Already been entered and
                CALL    #BB9F           ;is also signal to write in transparent mode.
                CALL    #BB93           ;Find current ink
                LD      (INK1),A        ;save it.
                LD      A,(IY+03)       ;Get our ink
                CALL    #BB90           ;Change to it.
                POP     HL              ;Get csr position back.
                JP      REINIT
```

```
SKIP:           LD      A,(IY+05)       ;Character that was in sprite pos.
                CALL    #BB5D           ;Put it back on screen.
                LD      A,#FE           ;Transparent mode flag.
                CALL    #BB9F           ;Send it.
                CALL    #BB93           ;Current ink
                LD      (INK1),A        ;Save it
                LD      A,(IY+03)       ;Our ink from sprite table.
                CALL    #BB90           ;Make sure we write in it.
                POP     HL              ;Csr posit our routine.
                LD      A,(IY+02)       ;Direction byte
                CP      #01
                JR      Z,LEFT
                CP      #02             ; Left Diagonally & up ?
                JR      Z,LDIAU
                CP      03
                JR      Z,UP

                CP      #04             ; Right diagonally & up ?
                JR      Z,RDIAU
                CP      #05             ; Right ?
                JR      Z,RIGHT
                CP      #06             ; Right diagonally & down ?
                JR      Z,RDIAD
                CP      #07             ; Down ?
                JR      Z,DWN
                CP      #08             ; Left diagonally & down ?
                JR      Z,LDIAD
                RET                     ; Return if non of these.
        LEFT:   DEC     H
                JR      RESTOR          ; Decrement column position.
                RIGHT:  INC     H
                JR      RESTOR          ; Increment column position.
                UP:     DEC     L
                JR      RESTOR          ; Decrement Row position.
        DWN:    INC     L
                JR      RESTOR          ; Increment Row position.
        LDIAU:  DEC     L
                DEC     H
                JR      RESTOR
        LDIAD:  INC     L
                DEC     H
                JR      RESTOR
        RDIAU:  DEC     L
                INC     H
                JR      RESTOR
         RDIAD: INC     L
                INC     H
RESTOR:         LD      A,H             ;Get X pos
                CP      01              ;1st x pos
                JR      C,REINT2        ;JP if less than
                CP      41              ;Last X pos+1
                JR      NC,REINT2       ;JP if greater than
                LD      A,L             ;Now do same for Y pos.
                CP      #01
                JR      C,REINT2
                CP      26
                JR      NC,REINT2
                LD      (IY+00),L       ;Save new X pos
                LD      (IY+01),H       ;Save new Y pos
```

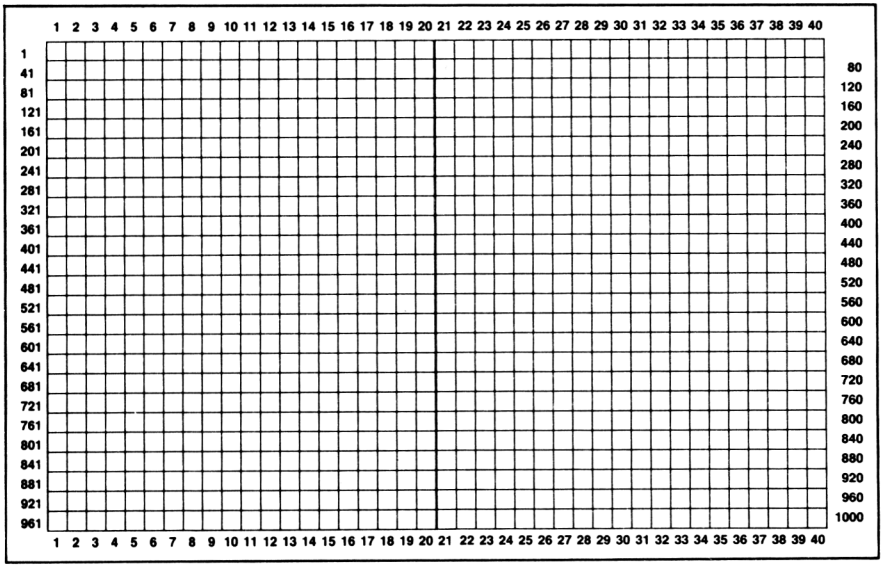
```
REINIT:     LD      HL,(IY+00)      ;Get our new positiion
            CALL    #BB75           ;Move csr to it
            CALL    #BB60           ;so we can read char on screen.
            LD      (IY+05),A       ;and save it in sprite table.
REINT2:     LD      HL,(IY+00)
            CALL    #BB75
            LD      A,(IY+06)       ;Get sprite character.
            CALL    #BB5D           ;write it to screen
            LD      HL,(BASPOS)     ;Get Basic's position
            CALL    #BB75           ;Mve csr to it
            LD      A,#00           ;Flag for opaque mode.
            CALL    #BB9F           ;Let Basic know we are now writing opaque.
            LD      A,(INK1)        ;Basic's ink.

            CALL    #BB90           ;Send it.
            RET                     ;JP back to main program.
        ;
        ;
FINDISP:    LD      A,(SPRNO)       ;Get sprite
            DEC     A               ;ALIGN  *SEE NOTES
            RLCA                    ;*2
            RLCA                    ;*4
            RLCA                    ;*8
            LD      C,A
            LD      B,#00
            LD      IY,SPRTBL
            ADD     IY,BC           ;IY now points to correct entry.
            RET
            ;
            ;
CRSPR:      LD      A,(IX+08)       ;Sprite no
            CP      09              ;Check if legal
            RET     NC              ;because only 8 allowed.
            LD      (SPRNO),A       ;save it for FINDISP
            CALL    FINDISP         ;find start in table for this sprite number.
            LD      A,(IX+02)       ;X pos
            LD      (IY+00),A       ;save in our table.
            LD      A,(IX+04)       ;Y pos
            LD      (IY+01),A
            LD      A,(IX+00)       ;Colour
            LD      (IY+03),A
            LD      A,(IX+06)       ;Pattern
            LD      (IY+06),A
            LD      A,(SPRNO)
            LD      (IY+07),A
            LD      A,#00           ;FLAG
            LD      (IY+04),A
                    RET                     ;Return  to  main program.
            ;
            ;
 SCRN:      LD      A,(IX+00)       ;X pos
            LD      L,A             ;
            LD      A,(IX+02)       ;Y pos
            LD      H,A
            LD      A,H             ;padding.
            CP      01
            RET     C
            CP      41              ;We've been through this before.
            RET     NC
            LD      A,L
            CP      01
            RET     C
```

```
                CP      26
                RET     NC
                CALL    #BB75           ;Update csr
                CALL    #BB60           ;Read screen
                LD      (PEEK),A        ;save it so we can peek it.
                RET                     ;That's all folks.
                ;
                ;
BASPOS:         DEFW    0000
INK1:           DEFB    00
BUF:            DEFS    #04
SPRNO:          DEFB    00
SPRTBL:         DEFS    64
PEEK:           DEFB    00
;
BLANK:          CALL    #BB78           ;Csr pos
                LD      (BASPOS),HL     ;Save it
                LD      HL,(IY+00)      ;Our position
                CALL    #BB75           ;Send it
                LD      A,(IY+05)       ;Basic's character
                CALL    #BB5D           ;Send it to screen
                LD      A,#FF           ;Sprite not created flag
                LD      (IY+00),A       ;Put it in sprite table
                XOR     A               ;Zero A reg
                LD      (IY+04),A       ; Flag
                LD      HL,(BASPOS)
                CALL    #BB75           ;Make sure csr back to basic's
                RET
                END
```

If we take advantage of the fact that we can create new commands for Basic, we can add useful routines as they take our fancy. New routines can be added to an existing utility simply by adding the new names to the **Name Table**, the addresses to the **Jump Table**, and tagging the new listing to the end of the existing program.

Some versions of Basic provide the command **PRINT@,nn** where **nn** is a **screen address**. This provides an easy way of formatting the screen when using lots of text. **Assembly Listing 3** provides this facility on the CPC464. The program also incorprates two other useful commands: **PEEK & POKE** to the screen.

The Amstrad's screen has 1000 print positions (25 lines x 40 columns), which would normally be addressed by using the **LOCATE x,y** command. Once the new program is installed each print position can be accessed by simulating the **PRINT@** command. The syntax for our new command is: **l.,<address>** and a Basic line would be similar to this, **10 l.,41:?"This is using a new command at position 41."** When using this new command, the screen is mapped as **Dia 8.**



To print a message starting at location **41** the Basic statement would be:-

10 l.,41:?"This starts at location 41.". The address can be a number in the range **1 - 1000** or a variable which will allow you to format neat displays. The following statements are all that you need to keep track of the cusrsor position, and easily control the printing zones.

CR = Cursor position.

CURSOR UP:	**CR = CR + 40 *(CR-40>0)**
CURSOR DOWN:	**CR = CR - 40 *(CR+40<1000)**
CURSOR FORWARD:	**CR = CR - (CR+1<1000)**
BACKSPACE CURSOR:	**CR = CR + (CR-1>0)**
MOVE TO TOP OF SCREEN: (SAME COLUMN)	**CR = CR-INT(CR/40)*40**
MOVE TO BOTTOM OF SCREEN (SAME COLUMN)	**CR = CR-INT(CR/40)*40+960**
CURSOR TO BEGINNING OF SAME LINE	**CR = INT(CR/40)*40**
CURSOR TO BEGINNING OF NEXT LINE.	**CR = -((CR>=960)*CR)-(CR<960)* (INT(CR/40)*40+40)**
CURSOR TO BEGINNING OF PREVIOUS LINE	**CR = -((CR<40)*CR)-(CR>=40)* (INT(CR/40)*40-40)**

```
10 CLS:CR = 1

20 l.,CR:?"Line 1,Column 1"

40 CR = CR -INT(CR/40)*40+960

50 l.,CR:?"Line 25,Column 1"
```

Once you have grasped the idea, using this method of formatting print statements is far superior to using the

LOCATE x,y statement - you can always check the cursor and move it to whatever location you desire by using the above formulas.

IVPOKE,<ADDRESS>,Value

Value is in the range 0 to 255 and corresponds to Ascii control characters,normal charcaters, and graphic blocks.

IVPOKE allows you to poke values to the screen. Screen mapping is exactly as that used in the previous command, and addresses lie in the range **1 to 1000**. To gain an insight into the advantage of this command try the following demonstration.

```
10 CLS

20 LOCATE 25,40:PRINT "K";

30 GOTO 30
```

NOW TRY THIS:

```
10 CLS

20 IVPOKE,1000,122

30 GOTO 30
```

Did you notice the difference ? The screen didn't scroll! This provides a useful way of writing to the screen without causing the screen to roll upward.

```
10 CLS

20 !.,1:?"TEST"

30 !VPOKE,1000,122

40 PRINT " VPOKE"

50 GOTO 50
```

Notice in the last demonstration that the cursor position remained at the **last print position** even though we poked the screen at the very last location.

!VPEEK,<address>,@PK

This command works in a similar way to the !POKE we used in an earlier chapter, it allows you to peek at the screen in any of the locations **1 to 1000.** The value of the PEEK is transferred into variable **PK.**

```
10 DEFINT P:PK=0

20 CLS

30 !.,500:?"XYZ"

40 !VPEEK,500,@PK

50 PRINT PK

60 !VPEEK,503,@PK

70 PRINT PK
```

When using this statement the variable **@PK must be initialised** to integer format, and **must have been defined (PK=0)** at the start of the program, or at least before the !VPEEK is used.

The assembly listing

The program starts by initialising the new commands and then **logging on** to the Basic Interpreter. Once this has been done the program exits back to Basic and the routines lie dormant until required by the Basic program.

IVPEEK uses two parameters, **address, and @PK.** When this section of the code is called, the parameters are passed to it with the **IX register** pointing to the last parameter, which in this case is **@PK.** Each parameter uses two bytes so, (IX+00) & (IX+01) hold the value of the last parameter (
PK), and (IX+02) & (IX+03) hold the value of the address. One important point to note with this routine is, **whenever a variable in the form variable (@PK) is passed to a program, it is the ADDRESS OF THE VARIABLE that is passed, NOT the value of the variable.** This means that once your extension routine has the variable's address it can change the value according to circumstances, and on returning to Basic, the variable will then contain the new value.

Once IVPEEK has the screen address it saves the current cursor location then aligns the cursor to the IVPEEK position by calling the **Divide** routine to convert the address to X,Y positions. The screen is then read and the character value is placed in variable PK's memory address. Before the routine returns to Basic, the original cursor position is restored.

IVPOKE works in exactly the same manner as **IVPOKE** except that a character is written to the screen. The character maybe any valid Ascii or graphic code in the range 0 to 255.

The **Divide** routine is useful for interfacing with other programs and an explanation of how it works will allow you to gain an insight into how you can use it within your own programs.

Divide works by successive subtraction - Chapter Two stated that another way to divide was to subtract the divisor as many times as possible before a negative value is encountered.

The D&E registers are loaded with 40_D which is equal to the number of print positions on one screen line. Because the first screen location is **1,1 (LOCATE 1,1)** the **B** register is loaded with an offset of **1** which aligns the final answer to the correct position. **OR A** clears the **Flag** register so that a **M (minus)** can be detected when the HL register pair reaches a negative value. E.g

```
If              HL = 81D
Then after      SBC HL,DE
                  HL = 41    and the program loops back
to do it again. SBC HL,DE
                  HL = 1     at this point the M flag has
                             not been set so the program
                             loops back again ....
                SBC HL,DE
                  HL = -39
```

This sets the **M** flag and the program jumps to **EXIT** where ADD HL,DE restores HL to the remainder -39 +40 = 1. HL now contains the remainder (1) which is in fact the **X position**. The Rom routine expects to have H = X position and L=Y position so H is loaded from L and L is loaded from B which was incremented each time a subtraction took place that didn't result in the M flag being set. (Every time 40 is successfully subtracted from HL without the M flag being set means that the print position is incremented one line of 40 characters.)

If you want to run the utility from within a Basic program you must load each byte (2 digits) into Data statements and Poke them into memory starting at location 40000_D. The program is then initialised by **CALL 40000**. The three new commands can now be used

from the keyboard or from within a program. **MEMORY 39999_D must be the first statement within your Basic program** to prevent Basic overwriting the routines.

To use the program as a separate file simply type in **Assembly Listing 3** using an Editor/Assembler, assemble the completed listing and save the object file to tape. (Save the source file also, just in case you made a typing error.) Before loading the program into memory type **MEMORY 39999_D**, and after loading, invoke the routine by **CALL 40000_D**.

This program made extensive use of the IX register and it would be prudent at this stage to explain the use of this 16 bit register. The **IX & IY registers are 16 bit registers known as the Index registers**. They allow the programmer an easy way of **index**ing memory locations and can have a range of -128 to +127.

LD IX,$3C00_H$

IX -03 points here =======>	2FFD = CA_H
	2FFE
	2FFF = 32_H
IX+00 points here ========>	3C00 = FF_H
	3C01
	3C02
	3C03
IX+04 points here ========>	3C04 = $3F_H$

LD A,(IX+04) would result in the A register containing **$3F_H$**.
LD A,(IX-03) would result in the A register containing **CA_H**.
You should be able to see how handy these two registers are for accessing data from a table, and this is the reason that the Amstrad uses the IX register to point to data when passing parameters from Basic.

Chapter 9

Bits & Pieces

We have already discussed the virtues of logical operations in Chapter 3. In this chapter we shall take a look at ways of utilising them in our Basic programming. We shall also take a look at another useful command that is often neglected by Basic programmers, **DEF FN.**

There are quite a few Basic operations that are **Bit specific**, that is, they operate in different ways depending on which bit is set within the byte. This will become obvious when we write a program to give us 3 new commands for using the Programmable Sound Generator in a later chapter - all the sound registers operate in different ways depending on the bit pattern within the byte.

A good example of how Basic operates with bits is the **JOY(0)** statement, which returns an integer value depending on which movement has been made by the operator. If the joystick has been moved to the left the value returned will be 4. However, if the operator pressed the fire button at the same time, the value returned by JOY(0) will be 20.

TABLE 9.1

JOY(0)

BIT		JOYSTICK MOVEMENT
0	SET	UP
1	SET	DOWN
2	SET	LEFT
3	SET	RIGHT
4	SET	FIRE2
5	SET	FIRE1

From Table 9.1 we can see that if the value returned by the function is 2 then the operator has moved the joystick DOWN and Bit 1 will be set.$00010_B = 2_D$. By testing for other values it is possible to check for multiple movements of the joystick.

20 = joystick moved LEFT AND FIRE BUTTON PRESSED
5 = joystick moved UP AND LEFT.

Using logical operations it is a simple matter to check the actual status of the bits rather than checking the value returned.

The ability of the Amstrad to set,reset,and test any bit from Basic allows us to use very sophisticated programming techniques, and a look at the methods employed to manipulate bits is a worthwhile exercise.

TO TEST ANY BIT A IN A TWO BYTE INTEGER N

We would use the following formula:

```
IF N AND 2^A THEN GOTO 20 ELSE RETURN
```

To test Bit 4

IF N AND 2^4 THEN GOTO 1200

TO SET ANY BIT A IN A TWO BYTE INTEGER N

Formula:

N = N OR 2^A

If we wanted to set bit 7 our program line would be:

10 N = N OR 2^7

TO RESET ANY BIT A IN A TWO BYTE INTEGER N use the following:

N = N AND NOT 2^A

To reset Bit 9:

N = N AND NOT 2^9

Bit manipulation can be used to its full advantage when we wish to calculate the remainder of any integer value divided by the power of 2. The remainder of N/4 is calculated by: **N AND NOT -4.**

```
10 X = 345

20 X= X/16

30 PRINT "THE REMAINDER OF X/16 IS ";X AND NOT -16
```

We can also check if a number is odd or even by using the above formula.

```
10 INPUT "PICK A NUMBER";N

20 IF N AND NOT -2 THEN GOTO 40

30 PRINT "NUMBER IS EVEN ":GOTO 10

40 PRINT "NUMBER IS ODD ": GOTO 10
```

The same type of operations can be used to good advantage on strings.

```
TO TEST BIT N IN A ONE BYTE STRING X$

                    10 IF ASC(X$)AND 2^N THEN GOTO 10(


TO SET BIT N IN A ONE BYTE STRING X$

                          10 X$=CHR$(ASC(X$)OR2^N)

Example:  Set Bit 3 in X$

                           X$=CHR$(ASC(X$)OR 2^3)


TO RESET BIT N IN A ONE BYTE STRING X$

                         X$=CHR$(ASC(X$)AND NOT 2^N)
```

In the preceding string operations we have used the ASC function to return the integer values of the character stored in x$. After performing a set,reset, or test we then used the CHR$ to restore the value back to a one byte string.

```
10 CLS

20 INPUT X$

30 FOR I = 0 TO 7

40 PRINT "BIT NUMBER ";I;

50 IF ASC(X$)AND 2^I THEN PRINT "YES":GOTO 70

60 PRINT "NO"

70 NEXT
```

To set up more than eight conditional tests we first need to calculate how many conditions are required. We can then create a string to the length calculated by using the following formula:

String Length = INT(Number of Conditions/8)+1

The string can be set to all zeros by:

X$ = STRING$(String Length,0)

It is, of course, possible to set all the bits to ones by substituting 255.

X$ = STRING$(String Length,255)

Using this type of Bit manipulation a template of "Yes" "No" conditions can be created to match the conditions required within the program. With a string of maximum length you can create 255*8 on/off conditions !

Using the Bit manipulating functions can be further enhanced by including them in a **DEF FN** statement. Once you have initialised the function you can call it simply by using the function name. This saves a lot of typing and makes our programs easier to read.

```
DEF   FN   TEST(ARG$,ARG)   =   (ASC(MID$(ARG$,
INT(ARG/8+1))AND   2^(ARG- FN TEST(n$,n) INT(ARG
/8)*8))<>0
```

The above Function will test any bit **ARG** in string **ARG$**. Whenever we want to call on this function,once it has been defined, we just call it by name e.g

FN TEST(n$,n)

If we wanted to test BIT 6 in P$ we call the function and substitute the desired arguments.

FN TEST(A$,6)

The **FN** command is one of the most useful tools we have at our disposal. With a little imagination Function statements can be made to do absolutely anything, and the advantage of this command is: once you have tested a Function you can use it within as many programs as you wish.

Most of us have, at sometime, created or used a Hexidecimal conversion type program. Just consider the power of the following function.

```
10  DEF FN D!(H$) = INSTR("123456789ABCDEF",MID$
(A$,1,1))*4096+ INSTR("123456789ABCDEF",MID$(A$2,
1))*256 + INSTR("123456789ABCDEF", MID$(A$,3,1))
*16+INSTR("123456789ABCDEF",MID$(A$,4,1))
 20 INPUT X$
 30 PRINT X$;"HEX" = FN D!(X$);" DECIMAL"
 40 GOTO 20
```

APPENDIX ONE

USEFUL SYSTEM JUMP BLOCK ROUTINES

$BB06_H$

Scans the keyboard until an input is detected. Returns with character in A register.

```
PUSH AF          ;Save AF registers CALL 0BB06H
LD HL,BUFFER     ;Point HL at a Buffer and....
LD (HL),A        ;Store character in there.
POP AF
```

$BB1E_H$

Check if a key has been pressed.
A = KEY NUMBER
NZ=KEY PRESSED
Z =KEY NOT PRESSED

```
        LD A,171        ;Check for key 171
SCAN:   CALL OBB1EH
        JR Z,SCAN       ; Key not pressed so go and
                        check again.
```

$BB5D_H$

Write a character to the screen.
A = CHARACTER TO WRITE

You must save BC,DE,HL as these register are corrupted on exit from this routine.

```
PUSH BC
PUSH DE
PUSH HL
LD A,32          ;send a space to current cursor position.
CALL 0BB5DH
POP HL
POP DE
POP BC
```

$BB60_H$

Find which character is displayed at the current cursor position.
Returns with character in A register.

```
CALL 0BB60H
CP "T"
JR NZ,NO
```

$BB75_H$

Move cursor to a new screen position.
ENTRY CONDITIONS
H = COLUMN
L = ROW

AF & HL CORRUPT ON EXIT.

```
PUSH AF
LD HL,0202H
CALL 0BB75H
```

BB78$_H$

Get current cursor position.
Returns with : H = Column
L = Row

BB81$_H$

Turn cursor on

BB84$_H$

Turn cursor off.

BB90$_H$

Change ink colour.

ON ENTRY A = INK NUMBER
CORRUPTS AF & HL ON EXIT

```
LD A,2 ; Ink 2
CALL 0BB90H
```

BB96$_H$

Set paper colour.

ON ENTRY A = NUMBER OF INK
CORRUPTS A & HL ON EXIT

```
LD A,2   ; Ink 2
CALL 0BB96H
```

BBAE$_H$

Get the start of the USER DEFINED MATRIX TABLE.

A = CHARACTER NUMBER OF THE FIRST ITEM IN TABLE.
HL = ADDRESS OF FIRST BYTE IN TABLE.

BBC0$_H$

Move graphic cursor to new position.

ON ENTRY DE = X CO-ORDINATE.
HL = Y CO-ORDINATE.

AF,BC,DE.HL ALL CORRUPT ON EXIT.

BBDE$_H$

Set graphics pen to ink colour in contained in A register.

```
LD A,3   ; Ink 3
CALL 0BBDEH
```

BBE4$_H$

Set graphics screen paper to ink contained in A register.

```
LD A,1   ; Ink 1
CALL 0BBE4H
```

BBEA$_H$

Plot a point.
ON ENTRY DE = X CO-ORDINATE.
HL = Y CO-ORDINATE.

AF & BC CORRUPT ON EXIT.

BBF6$_H$

Draw a line from current graphic cursor position.

DE = X CO-ORDINATE OF ENDPOINT.
HL = Y CO-ORDINATE OF ENDPOINT.

BBFC$_H$

Print a character on the graphics screen at current cursor position.

ON ENTRY A = CHARACTER.

BC,DE,HL CORRUPT ON EXIT.
Character written with top left hand corner at the current graphic position.

BB0E$_H$

Select a screen mode.

```
LD A,1   ; Mode 1
CALL 0BB0EH
```

ON EXIT BC,DE,HL CORRUPTED.

$BC14_H$

CLS

ON EXIT AF,BC,DE,HL CORRUPTED.

$BC32_H$

Set ink to a new colour.

ON ENTRY:
A = INK NUMBER
B = FIRST COLOUR.
C = SECOND COLOUR.

ON EXIT AF,BC,DE,HL CORRUPTED.

$BC38_H$

Set Border colours.

ON ENTRY:

B = FIRST COLOUR
C = SECOND COLOUR.

ON EXIT AF,BC,DE,HL CORRUPTED.

$BC5F_H$

Draw a horizontal line.

ON ENTRY:

DE = X CO-ORDINATE START
BC = X CO-ORDINATE END.
HL = Y CO-ORDINATE END.

$BC62_H$

Draw a vertical line.

ON ENTRY:

A = INK NUMBER
DE = X CO-ORDINATE.
HL = Y CO-ORDINATE START OF LINE.
BC = Y CO-ORDINATE END OF LINE.

$BCD1_H$

Log on a new Basic command.

ON ENTRY:

BC = NEW COMMAND TABLE.
HL = POINTER TO A 4 BYTE BUFFER.

$BD2B_H$

Send a character to the line printer.

ON ENTRY:
A = CHARACTER TO SEND.
If Character sent to printer successfully Carry [C] Flag set.

```
LD A,41
SEND: CALL 0BD2BH
JR NC, SEND   ; if not sent try again.
```

$BD34_H$

Send data to PSG.

ON ENTRY:

A = REGISTER NUMBER.
C = DATA TO SEND (0-255)

APPENDIX TWO

INKS AND LUMINANCE VALUE

LUMINANCE	INK	LUMINANCE	INK
0	BLACK	13	WHITE
1	BLUE	14	PASTEL BLUE
2	BRIGHT BLUE	15	ORANGE
3	RED	16	PINK
4	MAGENTA	17	PASTEL MAGENTA
5	MAUVE	18	BRIGHT GREEN
6	BRIGHT RED	19	SEA GREEN
7	PURPLE	20	BRIGHT CYAN
8	BRIGHT MAGENTA	21	LIME
9	GREEN	22	PASTEL GREEN
10	CYAN	23	PASTEL CYAN
11	SKY BLUE	24	BRIGHT YELLOW
12	YELLOW	25	PASTEL YELLOW
		26	BRIGHT WHITE

==========

BRAINTEASERS

FOR THE

AMSTRAD CPC 464

ISBN 0 9465 7634 3

A collection of programs worthy of the title 'Brainteasers'. All of the programs exploit the graphics capabilities of the Amstrad CPC 464 and many of the items contain an IQ rating at their conclusion. This collection has already been published for other computers and has been reviewed as 'an enjoyable book – far from the run of the mill – fresh programs never seen anywhere else before.' All programs have full details of how to change the lines to help 'budding' programmers.

The Author

Genevieve Ludinski is an experienced programmer and technical author with her own software house specialising in educational material.

Publishing date: February 1985

Available from good bookshops or direct from

Phoenix Publishing Associates Limited
14 Vernon Road, Bushey, Herts WD2 2JL.

at £5.95 plus 55p post and packing

ADVANCED PROGRAMMING TECHNIQUES ON THE AMSTRAD CPC 464

This book is designed to open up whole new worlds of programming opportunities for Amstrad users.

Here you will find routines and facilities which will save you hours of unnecessary typing and help you speed up your own games and utilities. Full explanations are given of the techniques used to create these unusual programming aids which include how to store machine code programs in rem statements, arrays and strings!!

Highlights of this unique book are programs for:

Creating sprites

Developing a completely new sound system

Generating graphics

This book will help you get even more than you thought possible from your Amstrad.

THE AUTHOR

KEITH HOOK has spent fifteen years in computer studies, has written several books on the subject and is a regular contributor to PERSONAL COMPUTER NEWS.

ALSO FROM PHOENIX

THE AMSTRAD PROGRAM BOOK
Peter Goode

BRAINTEASERS FOR THE AMSTRAD
Genevieve Ludinski

BUSINESS PROGRAMMING ON YOUR AMSTRAD
Peter Jackson

PHOENIX PUBLISHING ASSOCIATES LTD

ISBN 0-9465-7632-7

9 780946 576326 90000

£7.95

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/