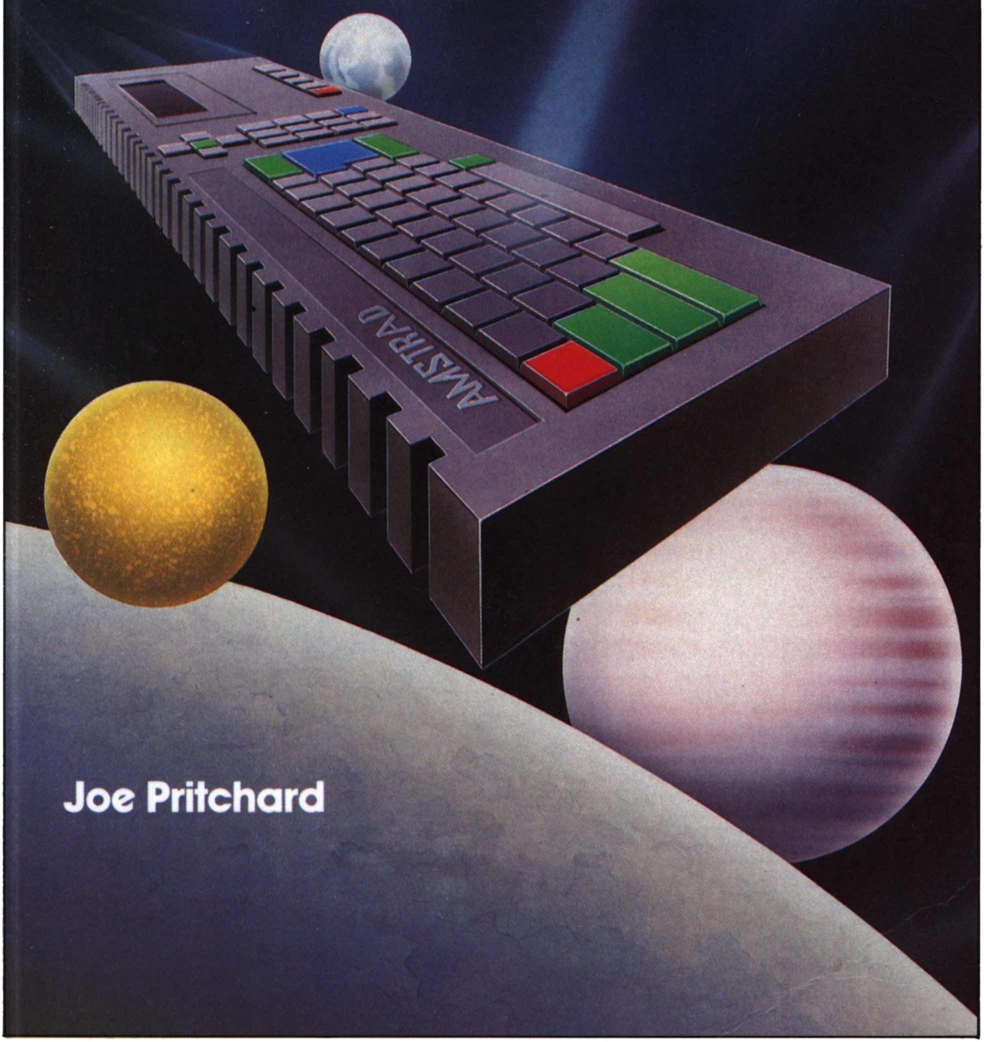
Ready Made Machine Language Routines for the Amstrad CPC464/CPC664



Joe Pritchard

Ready Made Machine Language Routines for the Amstrad CPC464/CPC664

Ready Made Machine Language Routines for the Amstrad CPC464/CPC664

Joe Pritchard

MELBOURNE HOUSE
PUBLISHERS

IN THE UNITED KINGDOM —
Melbourne House (Publishers) Ltd
Castle Yard House
Castle Yard
Richmond, TW10 6TF

IN AUSTRALIA —
Melbourne House (Australia) Pty Ltd
2nd Floor, 70 Park Street
South Melbourne, Victoria 3205

ISBN 0 86161 198 5

Printed and bound in Great Britain by
Biddles Ltd, Guildford and King's Lynn

Edition: 7 6 5 4 3 2 1
Printing: F E D C B A 9 8 7 6 5 4 3 2 1
Year: 90 89 88 87 86 85

Contents

Introduction

One of the daunting prospects that lies ahead of any machine code programmer is the writing of small routines for the machine of interest to perform particular tasks, such as printing strings of characters, saving areas of data on tape and inputting strings of characters from the keyboard. These routines often then turn up in all sorts of different programs, but the initial work of producing these routines can be a little tedious. The routines provided in this book will hopefully solve the problem. They have all been tested on the Amstrad CPC 464 Tape System, but should all work with the 664 machine or the standard machine with disks, although some relocation may be necessary in some cases. Screen routines given have been designed for use on a screen without windows, but this should give no problems. Wherever possible, routines will run equally well in all screen modes, and many can be used as useful subroutines for the BASIC programmer.

The programs were written using the ROM based MAXAM Assembler. I would like to thank my publishers for giving the idea the go ahead, and would like to acknowledge the help of Amsoft in the preparation of the book. Finally, I'd like to dedicate this book to my Mother and Father, for many years of tolerance beyond the call of duty!

Also, thanks to several local cats, who proved that it's possible to program a micro with a cat trying to attack you!

Joe Pritchard.
Nottingham, 1985

1.
Machine Language on the Amstrad

This book will provide you with a wide variety of ready-to-run machine language routines; however, a little knowledge about how machine code programs can be used on the Amstrad, and a few pointers about the structure of the Operating System, will allow you to get much more from your programming activities.

The Routines in this Book

The first thing to do, of course, is to actually get the programs listed into the computer. The best way to do this is to use an Assembler program, such as the Amstrad DEVPAC published by Amsoft. Alternatively, we can use the POKE command to enter the bytes that make up the machine code programs directly in to memory. In this book, you'll see that all the programs are listed in the below fashion:

Hexadecimal Byte			**Assembler Listing**	
2A	ØØ	ØØ	LD	HL,ØØØØ
CD	CØ	BB	CALL	&BBCØ
C9			RET	

thus allowing either method to be used. The Assembler listing can be typed into an assembler program and the hexadecimal bytes can be POKEd into memory in a variety of ways, the simplest being to use a simple BASIC loader of the sort listed below.

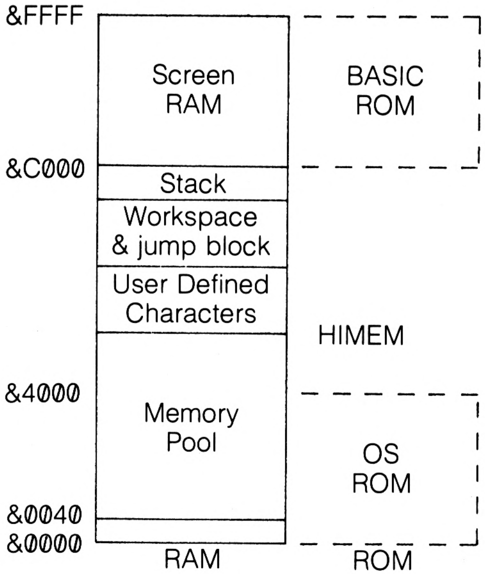
```
2Ø  FOR I=Ø TO n:REM n=number of bytes
3Ø  READ A$:POKE (address+I),VAL("&"+A$):REM address is
    where the code is to be loaded in memory
4Ø  NEXT I
5Ø  DATA DD,6E,ØØ,DD,66,Ø1,7E,DD,6E,Ø2,DD,66,Ø3,86,77,C9
```

The DATA statement contains the hexadecimal representations of the bytes that make up the program. Obviously, this method is rather useful for combining machine language routines with BASIC programs, as is often required. Listing the programs in the book as hexadecimal bytes will thus allow programmers without access to an Assembler to use the routines in the book.

Many of the routines will be written in such a way that they will be runnable at any address in RAM that is available to the programmer without alteration; such programs are said to be relocatable. Others require that they be run at a particular location in the memory; such routines are non-relocatable and alterations will be needed to allow these routines to run at addresses other than those given. However, details will be given wherever needed to allow you to make these alterations. It will thus be possible to select which of the routines listed in this book you require for a particular application, and load them into memory where you want them.

However, where in the memory of the computer can we actually store the machine code? If you've done a little machine code programming on the Amstrad already, you might like to skip this section and rejoin us when we look at the CALL routine. But, if you want a refresher course on the subject, here we go.

Memory Use on the Amstrad



Simplified Amstrad Memory Map

The area designated "Memory Pool" is the area of memory which is used to store our BASIC programs and variables. Any machine code programs that we write must be protected from being accidentally overwritten by BASIC, and they must be safe from the ravages of the rest of the Operating System, which uses various areas of memory as workspace. The way in which we create such a safe area of memory is quite simple; we "poach" some of the Memory Pool away from BASIC.

The last byte of RAM that is available to BASIC is given a special name — HIMEM. PRINT HIMEM at any time will give the address of the last byte of memory that is available to BASIC at that time. On turning on the system, this is 43903 or &AB7F. The byte at address 43904 is part of the definition of character number 240 in these circumstances, which is the first User Definable Character available to the user at turn on.

We "borrow" some of the memory normally available to BASIC by moving HIMEM down in memory, towards address &0000. The effect of this is to create space between HIMEM and the first byte of the User Defined Character definitions. We move HIMEM using the MEMORY command. For example,

```
MEMORY 39999
```

will set HIMEM to address 39999, thus allowing us to use the space between addresses 40000 and 43903 inclusive for machine language routines. This is adequate for most applications. There are a couple of points to note about the use of the MEMORY command. The first is that you should use SYMBOL AFTER to reserve space for as many User Defined Characters as you will require in your program BEFORE you use MEMORY to move HIMEM. The second point to note is that SYMBOL AFTER will not work properly whenever a file is open on the Cassette System. This is due to the fact that the act of opening a file causes HIMEM to be moved away from where SYMBOL AFTER expects to find it!

So, by using MEMORY we can reserve a safe area for our machine code programs, in to which we can POKE the bytes that make up the programs, as was mentioned earlier in the Chapter. We need now to be able to run the machine code. This is done with the CALL statement.

CALLing Machine Language Programs

You, as an Amstrad programmer, are rather lucky to have the CALL statement; most home computers have a very poor interface between BASIC and Machine Language. The Amstrad CALL statement is not terribly well documented in the "Amstrad CPC 464 User Instructions" but is a very versatile command indeed. As we'll be using it throughout the book to call our routines, let's examine it in some detail.

There are two different ways in which the instruction can be used, and examples of these two modes of operation are shown below.

```
CALL  &BD19
CALL  40000,A%,B%,C%
```

The first of these simply causes the machine language routine at address &BD19 of the Amstrad RAM to be executed. The second statement causes the routine at address 40000 to be executed, but in addition makes the current values of the three BASIC variables A%, B% and C% available to the machine language routine. This is clearly very useful. A% and the others are said to be the PARAMETERS of this call. Armed with this more advanced CALL statement, we can allow our machine code routines to interact directly with BASIC numeric and string variables.

So, let's take a closer look at this CALL with parameters. We will only consider routines that use Integer Numbers and Strings in this book; experience has shown that most applications that involve complex arithmetic and "Real" Numbers are best done in BASIC. This is partially due to the fact that such machine code programs take a long time to write, and are generally NOT as efficient as the routines present in the BASIC ROM to carry out complex arithmetic operations.

There are three broad classes of parameter that can be passed over as part of a CALL statement, and such a statement can have as many parameters as you can fit on a line (up to 255 characters)! Of course, the parameters for a particular CALL statement can be from any of the three classes of parameter. The parameter types are:

(i) A number, such as 100,2 or 1000, an Integer Variable name, such as A% or an expression that evaluates to give an integer result. The value passed should be in the range 0 to 65535 although, as we shall see later, there are a couple of points to be wary of. If an Integer Variable is to be a parameter, such as A%, and has not been given a value when it is included in the CALL statement then the variable will be treated as holding the value 0.
An example of this type of CALL is

```
CALL  40000,A%
```

where 40000 is the address of the routine to be called and A% is the parameter.

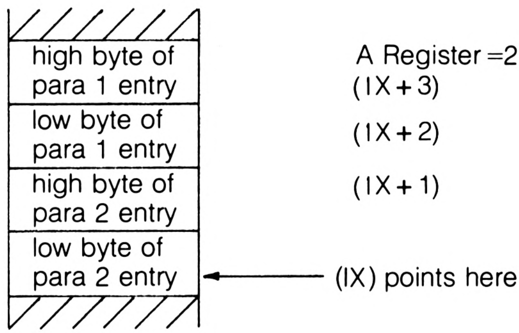
(ii) An Integer Variable name prefixed by the "@" character, such as @A%. This method of passing a numeric parameter to a machine code routine also allows us to have a "two way" transfer of information, to and from the machine language program, as we'll soon see.

(iii) A String Variable name prefixed by "@". This is the only way in which strings can be passed to machine code routines, We cannot pass String Constants, such as "Joe", to a routine, only the variables.

So, how do we gain access to the parameters passed over to the machine code programs? On entry to the routine at the address called, 2 of the CPU registers have been set up by the BASIC Interpreter to help you gain access to the parameters passed over. The A Register holds the number of parameters passed over, and the IX Index Register points to an area of memory called the PARAMETER BLOCK, which contains a two byte entry for each parameter passed to the machine code routine. IX points to the parameters in the below fashion, the exact contents of each two byte entry depending upon the type of parameter.

CALL 40000,para1,para2

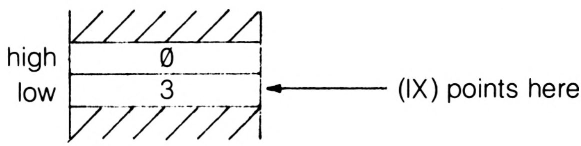
generates:



Let's now examine the contents we can expect to find in each parameter block entry for each type of parameter.

Integer Variables or Numbers

These include such things as A% or 5, and the parameter block entry will hold a two byte binary representation of the number. Thus for CALL 40000,3 the parameter block entry would be:



A word of caution is in order here; the two calls

```
CALL 40000,-1
CALL 40000,65535
```

will both leave &FFFF as the parameter block entry, this being the Two's Complement representation of −1 which is the way in which negative integers are stored internally in the Amstrad. Also, attempting to pass a value such as 65535 over to a machine code routine in an Integer Variable, such as

```
A%=65535:CALL  40000,A%
```

will cause the "Overflow" error, as integer variables can only hold numbers in the range −32768 to +32767. Retrieving these values from the parameter block into a Register or Register pair is quite simple.

```
LD   L,(IX+0)   ; low byte
LD   H,(IX+1)   ; high byte
```

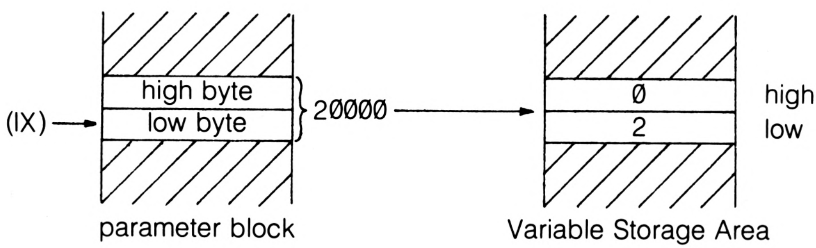
Variables Prefixed with @

So far, communication between BASIC and machine language has been a rather one way affair, with us being able to pass values from BASIC to machine code but not the other way around. Use of "@" allows us this facility. The parameter block entry for such a parameter is no longer the actual value of the variable, as it was for A% or 5, but is now the **address** of the variable; i.e. where in memory BASIC stores the variable.

For example,

```
CALL  40000,@H%
```

will generate a parameter block containing the address of the H% variable in RAM. For the sake of argument, let's say the two byte parameter block entry holds the value 20000. This would indicate that the low byte of the value of H% is stored at address 20000 and the high byte of the value is stored at address 200001. Diagramatically, this can be shown as:



The useful thing about all this, of course, is that by altering the values held in the two bytes used to store the value of H%, we can alter the value of H% from our machine language program. Thus in this example, to set H% to a value of 7 we would load 7 into location 20000 and 0 into location 20001. As a short demonstration, examine the below program which is called with CALL 40000,@A%,n.

```
40000
DD 7E 00 LD   A,(IX+0)  ; get the value of n into A register
DD 6E 02 LD   L,(IX+2)  ; get address of the integer variable
DD 6E 03 LD   H,(IX+3)  ; in to the HL pair
77       LD   (HL),A    ; set the low byte to what is in
C9            RET       ; A and return to BASIC.
```

A% should be initialised to 0 before the call is made, as the program only affects the low byte of the variable in memory. n should thus be a value between 0 and 255. Of course, any integer variable can be used, not just A%. Thus

```
joe% =0:CALL 40000,@joe%,6
fred% =0:CALL 40000,@fred%,67
```

are both legal. In the first case, joe% will be set to hold the value 6 after the call and in the second case fred% will hold the value 67. n can also be a variable but without the @ prefix.

One final point about the use of @. It gives the user access to the address of a particular variable PROVIDED THAT the variable in question has been previously used, even if it's just been set to 0. If you attempt a call using @ with an unspecified variable, an error message will be given by the Amstrad. The reason for this is obvious; if a variable hasn't been previously used, then the BASIC Interpreter won't have an address that it can put in to the parameter block!

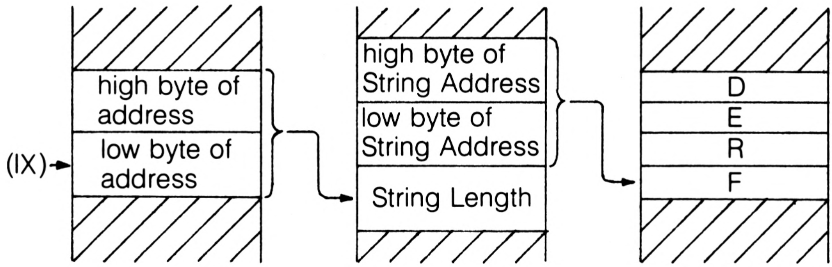
Passing Strings

Whereas numeric constants, such as "1" or "4000" can be passed to machine code programs, you cannot pass string constants such as "Help" or "Amstrad" to a machine code routine. You can pass string variables over, but the variable name must be prefixed by the "@" character.

Again, the string variable must be initialised before being used in a CALL statement to allow the BASIC Interpreter to work out where in memory the string is. A legal CALL involving a string variable is:

```
A$="fred":CALL 40000,@A$
```

The two byte parameter block entry for this type of parameter can be interpreted in the below fashion. It points to an area of memory called a string DESCRIPTOR BLOCK, which provides us with useful information about the string.



Parameter Block DESCRIPTOR STRING

The Descriptor thus tells us the length of the string and it's whereabouts in memory.

Of course, POKE can be used to place values in memory locations where they can be picked up by your machine code routines, and PEEK can be used to get values back from machine language into BASIC. However, CALL gives us a method that is both elegant and efficient in all but the simplest of cases, and so you'll see it often used in this book.

We'll now go on and look at a few hints about using machine language routines on the Amstrad, before looking at the use of ROM OS calls.

(i) Use of the Alternate Register Set

Under no circumstances should you use the alternate register set of the Z-8Ø. These registers are all used by the OS for various purposes and so it would be ill advised to alter the contents of any of these registers as the OS may require them at any time.

(ii) RAM overlaying ROM

By a clever feat of design, the ROM containing the OS has been overlaid with RAM which we can use for storage of our BASIC or machine code programs. Similarily, the BASIC ROM is overlaid by screen memory. Under normal circumstances, the RAM will be accessible to the programmer; if you want to read from ROM you have to be rather clever about it.

(iii) State of CPU Registers

Use of ROM routines will tend to leave various registers corrupted, and in many cases the Flag Register will also be corrupt. Thus any registers that you want to preserve should be Stacked using PUSH before calling ROM routines.

(iv) The CALL command

It has been suggested that on entering your machine code routine the registers should be preserved anyway to help the return to BASIC from the routine called with the CALL command. I've found that the machine recovers very well from CALL without this.

Use of ROM calls

We will use the Amstrad ROM routines for such tasks as writing to the screen and reading the keyboard. All these routines can be accessed by calling routines at particular RAM addresses using the Z-8Ø CALL instruction from our machine code programs. These are set up by the OS at reset or power on. These addresses are in an area of memory called the JUMPBLOCK, and all calls to ROM routines should be via these jumpblock entry points for the following reasons.

The first reason, which will be explored in more detail below, is that the ROMs are usually disabled, and so to use any ROM routines we must first enable the ROM. This is done automatically by the Jumpblock entry for a particular routine. Secondly, these Jumpblock addresses are guaranteed by Amstrad, no matter how many different versions of the OS there are. Thus a call to address &BB5A will always pass control to the OS routine that prints a character, no matter what version of the OS is in the machine. This effectively "future proofs" the machine, by ensuring that future alterations to the OS will not invalidate programs already written on earlier OS versions.

Thirdly, we can alter the Jumpblock for a particular OS routine and so alter the way in which the OS reacts to particular occurrences.

The Jumpblock

This is quite a large area of RAM just above the Memory Pool that holds the entry points for all the important OS routines and some of the BASIC ROM routines. A typical entry in the Jumpblock is shown below.

Address in RAM	Value	
&BB18	&CF	
&BB19	&56	low byte of address
&BB1A	&9B	high byte of address

&CF is a rather special code on the Amstrad; put simply, it specifies a jump to the routine that is held in ROM at the coded address given in the two bytes following it. The two byte address, in this example &9B56, is coded in the following fashion.

Bit 15 This bit of the address controls the BASIC ROM; if it is set to 1 then the BASIC ROM is disabled. If set to Ø the the ROM is enabled.

Bit 14 This bit of the address controls the OS ROM. Again, if it is set to 1 the OS ROM is disabled and if set to Ø the OS ROM is enabled.

Bits Ø to 13 of the address specify the address of the routine within the ROM that has been selected by the upper two bits. As a concrete example, let's write the value of &9B56 in binary.

```
1 Ø Ø 1   1 Ø 11   Ø 1 Ø 1   Ø 1 1 Ø
Bit 15                   Bit Ø
```

Here, Bit 15 is set to 1, thus disabling the BASIC ROM. Bit 14 is set to Ø, and so the routine is in the OS ROM, which is enabled by this bit. The address held on Bits Ø to 13 is &1B56, and so this is the address of the routine within the OS ROM that will be called by a CALL to the Jumpblock entry at &BB18. &CF isn't the code for a proper Z-80 instruction, it's more of a "pseudo operation" that has been put together by Amstrad for this particular application.

We'll see later in the book how we can alter Jumpblock entries to augment or replace the usual OS functions. If you do this, the &C3, the code for JP, will replace the &CF, and the next two bytes will form the RAM address of the routine which you have written. Thus an entry in the Jumpblock of

```
&C3
&ØØ
&AØ
```

will cause a jump to a routine at address &AØØØ. The routine should then end in one of two ways. The first is with a RET instruction, which will cause the routine that you've added to replace totally the normal OS function. The second way is to end your program with the original contents of the Jumpblock, starting with the &CF byte. Note that patching the OS in this way may cause problems with later versions of the Operating System. In this latter case, your routine will be executed first and then control will be passed on to the usual OS routine.

And that, as they say, is that! The rest of the book is what you bought it for; ready made and documented machine code routines for the Amstrad. I hope that you will also experiment with the routines, and possibly develop them into routines that are even more versatile.

2.
Text Output Routines

In BASIC, text output is an easy job, as we have the PRINT command to do all the work for us. This isn't so in machine code, so we have to put together a few routines to enable us to do such mundane jobs as print messages and numbers to the screen. We'll see a variety of these in this Chapter.

The first is called SPRINT, which stands for String PRINT, and it allows us to print messages to the screen.

SPRINT

Prints a string to the currently selected stream in the current text colour to the specified screen position. The routine is relocatable. Control Codes are PRINTED, not acted upon.

Entry Requirements: B holds X position
C holds the Y position
IX points to the string in memory. The string must end with a CHR$(Ø)

Exit Conditions: All corrupt.

Length: 37 Bytes.

```
DD E5      SPRINT  PUSH  IX
CD 54 BB           CALL  &BB54      ; enable text on this stream
3E 1F              LD    A,&1F
CD 5A BB           CALL  &BB5A      ; position text cursor
78                 LD    A,B        ; using CHR$(31)
CD 5A BB           CALL  &BB5A
79                 LD    A,C
CD $A BB           CALL  &BB5A
DD E1              POP   IX
DD 7E ØØ   LOOP    LD    A,(IX+Ø) ; get char of string
FE ØØ              CP    ØØ         ; is it zero?
```

```
C8              RET    Z        ; if so, finished
DD E5           PUSH   IX
CD 5D BB        CALL   &BB5D    ; print character
DD E1           POP    IX
DD 23           INC    IX       ; point to next char.
18 EF           JR     LOOP     ; round again
```

Notes If a sentence or message is too long to fit on the line on which it started, it will continue on the next line.

The best way to see this program in action is to type in the demonstration program that is listed below. There are a few additional machine code instructions in it, but these are simply to set the registers up before calling SPRINT. The additional instructions are

```
LD    IX,&9C4Ø ; address of string
LD    B,nn     ; X coordinate poked in
LD    C,nn     ; Y coordinate poked in
```

The values in the B and C registers when the SPRINT routine is called must be appropriate to the screen mode in use at that time. The BASIC program is:

```
  10 REM SPRINT Demonstration
  11 MEMORY 39999
  15 CLS
  2Ø GOSUB 1ØØØ
  3Ø LOCATE 1,2Ø
  4Ø INPUT "X Position";x
  5Ø INPUT "Y Position";y
  6Ø INPUT "String";a$
  7Ø FOR I=1 TO LEN(a$): POKE (39999+I),ASC
     (MID$(a$,I)):NEXT I
  8Ø POKE (39999+I),Ø:REM terminate the string in memory
  9Ø POKE 4Ø2Ø5,x:POKE 4Ø2Ø7,y:REM POKE in x and y
 1ØØ CALL 4Ø2ØØ
 111 GOTO 3Ø
1ØØØ REM subroutine pokes in machine code
1ØØ5 FOR I=Ø TO 44
1Ø1Ø READ a$: POKE (4Ø2ØØ+I),VAL("&"+a$)
1Ø2Ø NEXT I
1Ø3Ø RETURN
2ØØØ DATA dd,21,4Ø,9c,Ø6,ØØ,Øe,ØØ:REM m/c to set registers
     up
2Ø1Ø DATA dd,e5,cd,54,bb,3e,1f,cd,5a,bb,78,cd,5a,bb,79,
     cd,5a,bb,dd,el,dd,7e,ØØ,fe,ØØ,c8,dd,e5,cd,5d,bb,dd,
     el,dd,23,18,ef
```

Running this program will allow you to position a string on the screen from machine code.

You may have noticed in the above routine that we use a control code, in this case ASCII code 31, to position the text on the screen. The Amstrad Manual shows the various control codes available to us, and documents their effects. We'll now look at a short routine that enables us to use these control codes from our programs so that we can take full advantage of the facilities offered by the Amstrad machine.

The list, by the way, can be found on Pages Chapter 9 2-4 of the User Manual. A quick examination there will reveal many useful codes. In addition, of course, each control code has a printable character associated with it, and this printable character is the one printed out by SPRINT. Our next routine, CPRINT, prints nothing to the screen but EXECUTES the control codes; thus passing CHR$(7) to CPRINT will cause a "BEEP" to be generated, rather than printing the little "Space Invader" style character that SPRINT prints.

CPRINT

Sends a string of control codes to the text VDU, and the routine is relocatable.

Entry Requirements: B holds the number of characters. IX points to a block of memory holding the codes. The character code in the address pointed to by IX will be the first one sent.

Exit Conditions: AF,BC and IX are corrupt.

Length: 14 Bytes.

```
CD 54 BB  CPRINT  CALL  &BB54  ; enable text VDU
DD 7E ØØ  LOOP    LD    A,(IX+Ø)
CD 5A BB          CALL  &BB5A  ; send control code
DD 23             INC   IX
1Ø F5             DJNZ  LOOP   ; all done?
C9                RET          ; yes
```

As a short example of its use, the below example shows how we might use CPRINT to define a character — the machine code equivalent of the SYMBOL command. A block of bytes is set up in memory holding the data for the character. So, for SYMBOL 24Ø,1,3,7,15,31,63,127,255 we would set the below block up.

```
n+9    255    last byte of definition
       127
       63
       31
```

	15	
	7	
	3	
	1	
n+1	240	character to be defined
n+0	25	Control Code for Symbol.

CPRINT

```
10    MEMORY 39999
20    PRINT CHR$(240)
30    GOSUB 90
40    RESTORE 150
50    FOR i=0 TO 9 : READ a : POKE (40000+i),a : NEXT
60    CALL 40200
70    PRINT CHR$(240)
80    END
90    FOR i=0 TO 19
100   READ A$ : POKE (40200+I),VAL("&"+A$)
110   NEXT I
120   DATA 06,0a,dd,21,40,9c
130   DATA cd,54,bb,dd,7e,00,cd,5a,bb,dd,23,10,f5,c9
140   RETURN
150   DATA 25,240,1,3,7,15,31,63,127,255
```

Assume n=41000.

The machine code to call CPRINT would then simply be

```
LD      IX,41000
LD      B,10
CALL    CPRINT
RET
```

An examination of the Control Codes list will show you how to use CPRINT to change the text PEN and PAPER colour, giving you the chance to change colour when using CPRINT in conjunction with SPRINT.

So far we've only looked at positioning a text string at the text cursor. What about a machine code equivalent of the BASIC TAG command? GPRINT allows us to position text at the graphics cursor, and it also allows us to specify the spacing in screen pixels between the characters that make up the string.

GPRINT2 is an extended version of GPRINT that allows us to use an extended CALL command to make use of this routine from BASIC with greater ease.

GPRINT

Prints, in the current graphics colour, a string at the X and Y coordinates specified. The spacing between characters printed can also be specified. This routine is Relocatable.

Entry Conditions: BC holds number of pixels between characters
HL Y Coordinate
DE X Coordinate
IX points to the string in memory.

String must end with CHR$(Ø).

Exit Conditions: All Registers Corrupt.

Length: 37 Bytes.

```
C5        GPRINT PUSH  BC       ; preserve the registers
D5               PUSH  DE
E5               PUSH  HL
CD CØ BB         CALL  &BBCØ    ; position text cursor
                                ; at X,Y
E1               POP   HL       ; restore registers
D1               POP   DE
C1               POP   BC
DD 7E ØØ         LD    A,(IX+Ø); get character
FE ØØ            CP    ØØ       ; if zero, finish
C8               RET   Z
C5               PUSH  BC
D5               PUSH  DE
E5               PUSH  HL
CD FC BB         CALL  &BBFC    ; print char at the t
                                ; graphics cursor.
E1               POP   HL
D1               POP   DE
C1               POP   BC
E5               PUSH  HL       ; preserve Y pos.
D5               PUSH  DE       ; get the X position
E1               POP   HL       ; into HL reg.
AF               XOR   A        ; Clear Carry
ED 4A            ADC   HL,BC    ; update the X position
E5               PUSH  HL       ; return X to
D1               POP   DE       ; DE register
E1               POP   HL       ; get Y back
DD 23            INC   IX       ; get next char pointed to.
18 DB            JR    GPRINT   ; round again
```

GPRINT

```
5       MODE 1
10      MEMORY 39999
15      CLS
20      GOSUB 1000
30      LOCATE 1,20
60      INPUT "String":a$
70      FOR i=1 TO LEN(a$) : POKE (39999+i),ASC(MID$(a$,i)) : N
        EXT i : POKE (39999+i),0
80      CALL 40200
90      GOTO 30
1000    FOR i=0 TO 49
1010    READ a$ : POKE (40200+i),VAL("&"+a$)
1020    NEXT i
1030    RETURN
2000    DATA dd,21,40,9c,11,c8,0,21,c8,0,1,10,00
2010    DATA c5,d5,e5,cd,c0,bb,e1,d1,c1,dd,7e,00,fe,00,c8,c5,d5
        ,e5,cd,fc,bb,e1,d1,c1,e5,d5,e1,af,ed,4a,e5,d1,e1,dd,23,1
        8
2020    DATA db
```

Notes In this routine, any text that goes off the right hand edge of the screen is lost. The value in BC can be varied to suit your requirements, but the below values might give you good starting points.

Mode Ø   BC=32
Mode 1   BC=16
Mode 2   BC=8

Obviously, the values loaded into DE and HL as X and Y coordinates also depend upon the screen mode in use. The below program, GPRINT2, will allow you to experiment more easily with this routine by providing the instructions needed to get the parameters from a CALL statement in BASIC.

GPRINT2

This routine is called by

CALL address,x%,y%,b%,@a$

where x% =X Coordinate, y% =Y Coordinate, b% =Character separation and a$ holds the string to be printed. Of course, the three numeric parameters can also be constants. 'address' is the address to which you have loaded the routine.

Rather than repeat the listing of GPRINT, I'll list the extra instructions that are required to isolate the parameters, and then I'll give a BASIC program to demonstrate the routine. GPRINT2 is 71 bytes long, including GPRINT.

```
FE Ø4          GPRINT2CP    4          ; see if 4 parameters
CØ                    RET   NZ         ; if not, go back
DD 6E ØØ              LD    L,(IX+Ø); assemble the string
DD 66 Ø1              LD    H,(IX+1); address from the string
23                    INC   HL         ; descriptor block
4E                    LD    C,(HL)     ;
23                    INC   HL
46                    LD    B,(HL)     ; get the address into BC
C5                    PUSH  BC
DD 4E Ø2              LD    C,(IX+2)
DD 46 Ø3              LD    B,(IX+3); character spacing now in
                                       ; the BC pair
DD 6E Ø4              LD    L,(IX+4)
DD 66 Ø5              LD    H,(IX+5); Y coord in HL pair.
DD 5E Ø6              LD    E,(IX+6)
DD 56 Ø7              LD    D,(IX+7); X coord. now in DE pair.
DD E1                 POP   IX         ; string address in IX
```

The rest of the program is just GPRINT. Note that, just as in GPRINT, the string to be printed out must end in CHR$(Ø). The below BASIC program demonstrates the use of GPRINT2.

```
  10  MODE 2
  20  MEMORY 39999
  3Ø  GOSUB 1ØØØ: REM poke the machine code
  4Ø  LOCATE 1,22
  5Ø  INPUT "String"; $
  6Ø  a$=a$+CHR$(Ø):REM add the terminator character
  7Ø  INPUT "X Coordinate"; x%
  8Ø  INPUT "Y Coordinate"; y%
  9Ø  INPUT "Character Spacing"; b%
 1ØØ  CALL 4Ø2ØØ,x%,y%,b%,@a$: REM call the routine
 11Ø  GOTO 4Ø
1ØØØ  REM Routine to POKE in the bytes
1Ø1Ø  REM Note that if you change the address you
1Ø2Ø  REM will also have to change the address in line 1ØØ
1Ø3Ø  FOR I=Ø TO 7Ø
1Ø4Ø  READ a$:POKE ($Ø2ØØ+I),VAL("&"+a$)
1Ø5Ø  NEXT I
1Ø6Ø  DATA fe,Ø4,cØ,dd,6e,ØØ,dd,66,Ø1,23,4e,23,46,c5,dd,4e,Ø2
      dd,46,Ø3,dd,6e,Ø4,dd,66,Ø5,dd,5e,Ø6,dd,56,Ø7,dd,el
1Ø7Ø  DATA c5,d5,e5,cd,cØ,bb,el,dl,cl,dd,7e,ØØ,fe,ØØ,c8,c5,d5
      ,e5,cd,fc,bb,el,dl,cl,e5,d5,el,af,ed,4a,e5,dl,el,dd,
      23,18,db
```

Although we'll be taking a closer look at graphic operations in the next chapter, a short diversion is in order here in order to explain how we can change the graphics pen colour. I mention it here because text printed with GPRINT or GPRINT2 will be printed in the current graphics colour. A ROM routine will enable us to set the Graphics Pen Colour, and it is called GRA SET PEN.

GRA SET PEN

Changes the Graphics Pen Colour.

Entry Requirements: A = PEN Colour desired.

Exit Conditions: AF Corrupt.

Simply make a CALL to address &BBDE. Thus,

```
LD     A,1
CALL   &BBDE
RET
```

will set the graphics colour to PEN 1.

Printing Numbers

It's very easy in BASIC to print numbers out; we simply use PRINT. However, no such command exists in machine code, and so we must write our own routine to deal with the problem. Routines to print the contents of CPU registers as numbers can be very useful, whether it be for a utility program or simply printing the current score in a games program to the screen. We conclude this Chapter on Text output by looking at a variety of routines to perform the following tasks:

(i) Print the contents of the A Register as either a binary, hexadecimal or decimal number.

(ii) Print the contents of the HL Register pair as a binary, hexadecimal or decimal number.

Let's start with printing binary numbers out.

PBINA

Prints the contents of the A register as an 8 digit binary number to the screen at the current text position and in the current text colour. The routine is Relocatable.

Entry Requirements: A holds the number to be printed.

Exit Conditions: AF, BC Corrupt.

Length: 26 Bytes.

```
4F          PBINA   LD    C,A      ; copy A
Ø6 Ø8               LD    B,8      ; 8 bits in A register
```

```
CB 21      LOOP   SLA    C          ; shift bits to the
                                    ; left putting MSB in Carry
38 Ø9             JR     C,ZERO
3E 3Ø             LD     A,&3Ø      ; ASCII code for 'Ø'
C5                PUSH   BC
CD 5A BB          CALL   &BB5A      ; prints a 'Ø'
C1                POP    BC
18, Ø7            JR     OUT
3E 31      ZERO   LD     A,&31      ; ASCII code for '1'
C5                PUSH   BC
CD 5A BB          CALL   &BB5A      ; print it
C1                POP    BC
1Ø EA             DJNZ   LOOP       ; all bits shifted?
C9                RET               ; yes, so finish.
```

PBINA

```
10   MEMORY 39999
20   FOR i=0 TO 27
30   READ a$ : POKE (40200+i),VAL("&"+a$)
40   NEXT i
45   INPUT "Value";a
46   POKE 40201,a
50   CALL 40200
55   GOTO 45
60   DATA 3e,ff,4f,06,08,cb,21,38,09,3e,30,c5,cd,5a,bb,c1,18
     ,07,3e,31,c5,cd,5a,bb,c1,10,ea,c9
70   DATA 06,10,cb,25,cb,15,38,0b,3e,30,c5,e5,cd,5a,bb,e1,c1
     ,18,09,3e,31,c5,e5,cd,5a,bb,e1,c1,10,ea,c9
```

One useful application of this routine is to give you a look at the status of individual flags in the flag register. Obviously, the F register has to be copied in to the A register before this can be done but this is not too difficult.

The next routine prints out the contents of HL in a similar fashion.

PBINHL

This routine prints out the contents of the HL pair as a 16 digit binary number. Number is printed at the current text position and in the current text colour. The routine is relocatable.

Entry Requirements: HL holds the number.

Exit Conditions: AF, BC and HL Corrupt.

Length: 31 Bytes.

```
Ø6 1Ø       PBINHL  LD    B,16     ; 16 digits in number
CB 25       LOOP    SLA   L        ; shift all 16 bits one
CB 14               RL    H        ; bit to left, MSB into C
38 ØB               JR    C,ZERO
3E 3Ø               LD    A,&3Ø    ; ASCII code for Ø
C5                  PUSH  BC
E5                  PUSH  HL
CD 5A BB            CALL  &BB5A    ; print it
E1                  POP   HL
C1                  POP   BC
18 Ø9               JR    OUT
3E 31       ZERO    LD    A,&31    ; ASCII for 1
C5                  PUSH  BC
E5                  PUSH  HL
CD 5A BB            CALL  &BB5A    ; print '1'
E1                  POP   HL
C1                  POP   BC
1Ø E4               DJNZ  LOOP     ; if not Ø, go round again
C9                  RET
```

PBINHL

```
10    MEMORY 39999
20    FOR i=0 TO 33
30    READ a$ : POKE (40200+i),VAL("&"+a$)
40    NEXT i
45    INPUT "Value";a
50    high=INT(a/256)
60    low=a-(high*256)
80    POKE 40201,low
81    POKE 40202,high
90    CALL 40200
100   GOTO 45
110   DATA 21,00,00
120   DATA 06,10,cb,25,cb,14,38,0b,3e,30,c5,e5,cd,5a,bb,e1,c1
      ,18,09,3e,31,c5,e5,cd,5a,bb,e1,c1,10,e4,c9
```

It is, however, more common for us to want to print out the contents of a register in either decimal or hexadecimal notation. The next two routines deal with the printing out of number in hexadecimal representation.

PNUMA

Prints the A Register contents as a two digit hexadecimal number at the current text cursor and in the current text colour. The routine is Relocatable.

Entry Requirements: A holds the number.

Exit Conditions: AF, BC Corrupt.

Length: 41 Bytes.

```
Ø6 ØØ      PNUMA   LD     B,Ø       ; B used as a flag
4F                 LD     C,A
CB 1F              RR     A         ; next instructions move
CB 1F              RR     A         ; the 4 high order bits
CB 1F              RR     A         ; into low order part of
CB 1F              RR     A         ; A register
E6 ØF      PRINLO  AND    &ØF       ; mask them off
FE ØA              CP     &ØA       ; if greater than or equal
30 08              JR     NC,ATOF   ; 1Ø jump
C6 3Ø              ADD    A,&3Ø     ; make number in A
                                    ; into a character Ø to 9
C5                 PUSH   BC
CD 5A BB           CALL   &BB5A     ; print digit
18 Ø6              JR     OUT
C6 37      ATOF    ADD    A,&37     ; convert to character in
                                    ; range 'A' to 'F'
C5                 PUSH   BC
CD 5A BB           CALL   &BB5A     ; print it
C1         OUT     POP    BC
78                 LD     A,B       ; test flag, if 1 then all
FE Ø1              CP     1         ; the digits have been
C8                 RET    Z         ; printed, so return
79                 LD     A,C       ; get number back for
                                    ; second digit
Ø6 Ø1              LD     B,1       ; set flag to 2
18 E2              JR     PRINLO
```

PNUMA

```
5       REM PNUMA
10      MEMORY 39999
20      FOR i=0 TO 42
30      READ a$ : POKE (40200+i),VAL("&"+a$)
40      NEXT i
45      INPUT "Value for A Register";a
60      POKE 40201,a
90      CALL 40200
95      PRINT
100     GOTO 45
110     DATA 3e,00
120     DATA 06,00,4f,cb,1f,cb,1f,cb,1f,cb,1f,e6,0f,fe,0a,30,08
        ,c6,30,c5,cd,5a,bb,18,06,c6,37,c5,cd,5a,bb,c1,78,fe,01,c
        8,79,06,01,18,e2
```

At the heart of this routine are the two addition instructions that convert the values Ø to 9 into the corresponding codes and the digits A to F into their corresponding ASCII codes.

PNUMHL

Prints out the contents of the HL register paid as a hexadecimal number with 4 digits. The number is printed at the current text position and in the current text colour. The code is Non Relocatable, and the bytes given below are for loading at address 41ØØØ. However, see the below Notes to see how to alter the code.

Entry Requirements: HL holds the number.

Exit Conditions: AF, BC Corrupt.

Length: 51 Bytes.

```
7C          PNUMHL LD    A,H
CD 31 AØ           CALL  PNUMA
7D                 LD    A,L
CD 31 AØ           CALL  PNUMA
C9                 RET
Ø6 ØØ       PNUMA  LD    B,Ø
4F                 LD    C,A
CB 1F              RR    A
CB 1F              RR    A
CB 1F              RR    A
CB 1F              RR    A
E6 ØF       PRINLO AND   &ØF
FE ØA              CP    &ØA
3Ø Ø8              JR    NC,ATOF
C6 3Ø              ADD   &3Ø
C5                 PUSH  BC
CD 5A BB           CALL  &BB5A
18 Ø6              JR    OUT
C6 37       ATOF   ADD   &37
C5                 PUSH  BC
CD 5A BB           CALL  &BB5A
C1          OUT    POP   BC
78                 LD    A,B
FE Ø1              Cp    1
C8                 RET   Z
79                 LD    A,C
Ø6 Ø1              LD    B,1
18 E2              JR    PRINLO
```

PNUMHL

```
5       REM pnumhl
10      MEMORY 39999
20      FOR i=0 TO 6
30      READ a$ : POKE (40200+i),VAL("&"+a$)
40      NEXT i
41      FOR i=0 TO 49
42      READ a$ : POKE (41000+i),VAL("&"+a$)
43      NEXT i
45      INPUT "Value for HL register";a
60      POKE 40202,INT(a/256)
61      POKE 40201,(a-((INT (a/256))*256))
90      CALL 40200
95      PRINT
100     GOTO 45
110     DATA 21,00,00,cd,28,a0,c9
120     DATA 7c,cd,31,a0,7d,cd,31,a0,c9,06,00,4f,cb,1f,cb,1f,cb
        ,1f,cb,1f,e6,0f,fe,0a,30,08,c6,30,c5,cd,5a,bb,18,06,c6,3
        7,c5,cd,5a,bb,c1,78,fe,01,c8,79,06,01,18,e2
```

Notes The observant amongst you will have noted that PNUMHL is simply two calls to PNUMA. It is in this that the problems start with our relocatable code; because we have made part of our program into a subroutine, its address will depend upon the address to which the program has been loaded in memory. If PNUMHL is loaded to address N, then PNUMA will be found at address (N+9). Then, bytes (N+2) and (N+6) will have to be altered so as to hold the low byte of this address and bytes (N+3) and (N+7) will have to be changed to hold the high byte of the address of PNUMA.

The final number printing routines that we'll look at print the contents of registers out in decimal. Before we have a look at them, a little insight in to the algorithm used here will be useful.

It is simply a case of repeatedly subtracting a power of ten from the number being printed until the result is negative. We then add the power of ten to the result, restoring it to being a positive number. The number of times that particular power of ten was subtracted is the digit for that particular power of ten. This is then printed. The process is then repeated for the next lowest power of ten, and so on, until the units are subtracted, leaving the result Ø as the number being tested. Thus for an 8 bit register, we'd subtract, in turn, 1ØØ's, 1Ø's and finally units. These routines are useful in such applications as score tables in machine code games, printing line numbers in machine code games, printing line numbers in machine code utility routines and so on.

PDECA

Prints out the A register contents at the current text cursor position in the current text colour as a 3 digit decimal number. The routine is Non

Relocatable, but see the below Notes. The below bytes will run correctly at address 41060.

Entry Requirements: A holds the number.

Exit Conditions: AF,BC,DE are corrupt.

Length: 30 Bytes.

```
16 64       PDECA   LD     D,100
CD 70 A0            CALL   PDEC     ; prints no. hundreds
16 0A               LD     D,10
CD 70 A0            CALL   PDEC     ; print no. tens
16 01               LD     D,1
0E 00       PDEC    LD     C,0      ; zero the count
92          LOOP    SUB    D        ; subtract the power of ten
38 03               JR     C,OUT    ; if result negative, jump
0C                  INC    C
18 FA               JR     LOOP
82          OUT     ADD    A,D      ; restore A to positive
F5                  PUSH   AF
79                  LD     A,C
C6 30               ADD    &30      ; convert count to a digit
CD 5A BB            CALL   &BB5A    ; and print it
F1                  POP    AF
C9                  RET             ; done
```

PDECA

```
5       REM pdeca
10      MEMORY 39999
20      FOR i=0 TO 5
30      READ a$ : POKE (40200+i),VAL("&"+a$)
40      NEXT i
41      FOR i=0 TO 29
42      READ a$ : POKE (41060+i),VAL("&"+a$)
43      NEXT i
45      INPUT "Value for A Register";a
60      POKE 40201,a
90      CALL 40200
95      PRINT
100     GOTO 45
110     DATA 3e,00,cd,64,a0,c9
120     DATA 16,64,cd,70,a0,16,0a,cd,70,a0,16,01,0e,00,92,38,03
        ,0c,18,fa,82,f5,79,c6,30,cd,5a,bb,f1,c9
```

Notes If the routine above is loaded to an address other than 41060, say address N, then subroutine PDEC will be at address (N+12). Locations (N+3) and (N+8) must hold the low byte of address (N+12) and locations (N+4) and (N+9) must hold the high byte of address (N+12).

PDECHL

Prints out the contents of the HL register pair as a five digit decimal number, in the current text colour and at the current text cursor position. The routine is Non Relocatable. The below bytes will function correctly at address 412ØØ, and see the Notes for details about running the routine at other addresses.

Entry Requirements: HL holds the number.

Exit Conditions: AF, HL and DE Corrupt.

Length: 46 Bytes.

```
11 1Ø 27    PDECHL LD     DE,1ØØØ
CD ØB A1           CALL   PDECH  ; print no. of 1ØØØØ's
11 E8 Ø3           LD     DE,1ØØØ
CD ØB A1           CALL   PDECH  ; print no. of 1ØØØ's
11 64 ØØ           LD     DE,1ØØ
CD ØB A1           CALL   PDECH  ; print no. of 1ØØ's
11 ØA ØØ           LD     DE,1Ø
CD ØB A1           CALL   PDECH  ; print no. of 1Ø's
11 Ø1 ØØ           LD     DE,1
AF          PDECH  XOR    A      ; zero the counter
37          LOOP   SCF
3F                 CCF           ; clear carry flag
ED 52              SBC    HL,DE  ; do the subtraction
38 Ø3              JR     C,OUT  ; until it goes negative
3C                 INC
                          A
18 F7              JR     LOOP
19          OUT    ADD    HL,DE  ; restore to positive
C6 3Ø              ADD    &3Ø    ; convert count into
                                 ; ASCII digit and . . .
E5                 PUSH   HL
CD 5A BB           CALL   &BB5A  ; print it
E1                 POP    HL
C9                 RET
```

PDECHL

```
10      REM pdechl
20      MEMORY 39999
30      FOR i=0 TO 6
40      READ a$ : POKE (40200+i),VAL("&"+a$)
50      NEXT i
60      FOR i=0 TO 45
70      READ a$ : POKE (41200+i),VAL("&"+a$)
80      NEXT i
90      INPUT "Value for HL Register";a
```

```
100     POKE 40201,(a-(INT(a/256)*256))
110     POKE 40202,INT(a/256)
120     CALL 40200
130     PRINT
140     GOTO 90
150     DATA 21,00,00,cd,f0,a0,c9
160     DATA 11,10,27,cd,0b,a1,11,e8,03,cd,0b,a1,11,64,00,cd,0b
        ,a1,11,0a,00,cd,0b,a1,11,01,00,af,37,3f,ed,52,38,03,3c,1
        8,f7,19,c6,30,e5,cd,5a,bb,e1,c9
```

Notes In this case, the subroutine PDECH is at address &A10B. It will always be at address (N+27), where N is the address to which the whole program has been loaded. You will thus have to alter the subroutine calls at the start of this routine to point to the new address of PDECH if you load the routine to a different address to that given.

That completes this Chapter on text output, although we will see other routines that could have some bearing to text handling in later Chapters. We'll now go on to look at some graphics operations, including a variety of routines that will augment the usual BASIC commands for graphics handling.

3.
Graphics Routines

In this Chapter we'll see some routines that make use of the graphics abilities of the Amstrad. We'll also look at a few more text handling routines, such as routines to print large characters to the screen and some interesting routines to print "reversed" or "upside down" characters on the screen. In addition, there'll be a general purpose routine for drawing graphics shapes and pictures on the screen from a table of data, and several other useful graphics routines.

We'll start with some routines for printing large characters to the screen, useful for titles or demonstration programs. The first couple of these do not use graphics routines, but introduce some interesting ROM routines.

BIGCH

This prints a vastly enlarged character to the screen, 8 characters wide and 8 characters high. It will work in any screen mode, and the large character will be printed at the current text cursor position and will be in the current text colour. See later notes for relocation retails.

Entry Requirements: When used from BASIC, is entered by CALL address,n where n is the ASCII code of the character to be printed.
If called from another machine code routine, then A holds the value 1 and IX points to the character code to be printed.

Exit Conditions: All registers corrupt. Routine is exited when either the character has been printed OR the wrong number of parameters was passed to it.

Length: 80 bytes.

BIGCHAR

```
1000  MEMORY 39999
1010  SYMBOL 255,255,255,255,255,255,255,255,255
1020  GOSUB 1090
1030  CLS
1040  INPUT a$
1050  LOCATE 10,10
1060  FOR i=1 TO LEN(a$) : LOCATE i*8+1,10 : CALL 40200,ASC(M
      ID$(a$,i,1))
1070  NEXT
1080  END
1090   ASSEMBLE
1100  '         LIST
1110  '         ORG      40200
1120  '         CP       1
1130  '         RET      NZ           ; if not one parameter
1140  '                               ; return
1150  '         CALL     &B906        ; page in rom
1160  '         PUSH     AF           ; save ROM status
1170  '         LD       A,(IX)       ; get character to be
1180  '                               ; printed
1190  '         CALL     &BBA5        ; get pattern address
1200  '         LD       IX,40000     ; this is where the
1210  '                               ; pattern is to go
1215  '         LD       B,8          ; 8 bytes to move
1220  ' LOOP    LD       A,(HL)       ; HL holds pattern
1230  '                               ; address
1240  '         LD       (IX),A
1250  '         INC      HL
1260  '         INC      IX
1270  '         DJNZ     LOOP         ; transfer the 8 bytes
1280  '                               ; of the pattern
1290  '         POP      AF
1300  '         CALL     &B90C        ; restore ROM state
1310  '         LD       IX,40000
1320  '         LD       D,8          ; 8 bytes of character
1330  '                               ; definition
1340  ' LOOP1   LD       A,(IX)       ; get byte pointed
1350  '                               ; to by IX
1360  '         LD       C,A
1370  '         LD       B,8
1380  ' LOOP2   SLA      C            ; step through each
1390  '                               ; bit of byte
1400  '         JR       C,ZERO
1410  '         LD       A,32         ; print a space if
1420  '                               ; bit is '0'
1430  '         CALL     &BB5A
1440  '         JR       OUT
1450  ' ZERO    LD       A,255        ; print CHR$(255) if
1460  '                               ; bit is '1'
1470  '         CALL     &BB5A
1480  ' .OUT    DJNZ     LOOP2
1490  '         INC      IX
1500  '         LD       A,10         ; down 1 line
1510  '         CALL     &BB5A
1520  '         LD       B,8
1530  ' LOOP3   LD       A,8
1540  '         CALL     &BB5A
1550  '         DJNZ     LOOP3        ; send 8 CHR$(8)'s
1560  '         DEC      D
1570  '         JR       NZ,LOOP1     ; if all byte of
```

```
1580   '                                 ; definition have been
1590   '                                 ; done...
1600   '           RET                   ; finish
1610   RETURN
```

```
FE 01 C0 CD 06 B9 F5 DD 7E 00 CD A5 BB DD 21 40 9C 06 08
7E DD 77 00 23 DD 23 10 F7 F1 CD 0C B9 DD 21 40 9C 16 08
DD 7E 00 4F 06 08 CB 21 38 07 3E 20 CD 5A BB 18 05 3E FF
CD 5A BB 10 EE DD 23 3E 0A CD 5A BB 06 08 3E 08 CD 5A BB
10 F9 15 20 D5 C9
```

Notes In use, the routine can be called in a BASIC statement such as:

> y=1Ø:a$="string":FOR x=1 TO LEN(a$): LOCATE x*8+1,y:
> CALL 4Ø2ØØ,ASC(MID$(a$,x,1)):NEXT x

Obviously, I have assumed that the code has been assembled to address 4Ø2ØØ. The bytes in the above program listing are correct for this address. See the notes after VARCHAR for details about relocating these routines.

DCHAR

This routine prints out a character at double it's normal height. Again, it prints in any mode and will print to the screen at the current text cursor position in the current text colour.

Entry Requirements: When used from BASIC, it is accessed by a CALL address,n statement, where n is the ASCII code of the character to be printed.
If the routine is called from another machine code routine, then IX points to the character to be printed and A holds the value 1.

Exit Conditions: All registers are corrupt; the routine is exited when the character has been printed or when the wrong number of parameters have been passed in a CALL.

Length: 73 Bytes.

DCHAR

```
1000   REM dchar routine
1010   MEMORY 39999
1020   GOSUB 1070
1030   CLS
1040   A$="Joe.Pritchard.was.here"
```

```
1050  Y=10 : X1=3 : FOR I=1 TO LEN(A$) : LOCATE X1+I-1,Y : CA
      LL 40200,ASC(MID$(A$,I,1)) : NEXT
1060  END
1070  |ASSEM
1080  '          LIST
1090  '          ORG     40200
1100  '          CP      1
1110  '          RET     NZ            ; if not one parameter
1120  '                                ; return
1130  '          CALL    &B906         ; page in rom
1140  '          PUSH    AF            ; save ROM status
1150  '          LD      A,(IX)        ; get character to be
1160  '                                ; printed
1170  '          CALL    &BBA5         ; get pattern address
1180  '          LD      IX,40000      ; this is where the
1190  '                                ; pattern is to go
1200  '          LD      B,8           ; 8 bytes to move
1210  ' LOOP1    LD      A,(HL)
1220  '          LD      (IX),A        ; each byte is copied
1230  '          INC     IX            ; twice to give a 16
1240  '          LD      (IX),A        ; byte long double
1250  '          INC     IX            ; height definition
1260  '          INC     HL
1270  '          DJNZ    LOOP1
1280  '          POP     AF
1290  '          CALL    &B90C         ; restore the ROM status
1300  '          LD      A,254         ; define CHR$254 as top
1310  '          LD      HL,40000      ; of character
1320  '          CALL    &BBA8
1330  '          LD      A,255         ; define CHR$255 as
1340  '          LD      HL,40007      ; bottom half
1350  '          CALL    &BBA8
1360  '          LD      A,254         ; print top half
1370  '          CALL    &BB5A
1380  '          LD      A,10          ; down one line
1390  '          CALL    &BB5A
1400  '          LD      A,8           ; back space one
1410  '          CALL    &BB5A
1420  '          LD      A,255         ; bottom half
1430  '          CALL    &BB5A
1440  '          RET                   ; back to BASIC
1450  ' END
1460  RETURN
```

```
FE 01 C0 CD 06 B9 F5 DD 7E 00 CD A5 BB DD 21 40 9C 06 08
7E DD 77 00 DD 23 DD 77 00 DD 23 23 10 F2 F1 CD 0C B9 3E
FE 21 40 9C CD A8 BB 3E FF 21 47 9C CD A8 BB 3E FE CD 5A
BB 3E 0A CD 5A BB 3E 08 CD 5A BB 3E FF CD 5A BB C9
```

Notes This routine will work in all screen modes, but is particularly effective in Mode Ø, where the double width text is rendered more readable. If you wish to see the program in action, assemble the code to address 4Ø2ØØ and try the below lines of BASIC. The bytes above are for this address.

```
100 x=1:y=10:REM get screen position
110 MODE 2
```

```
120 FOR i=x TO LEN(a$):LOCATE x+i-1,y
130 CALL 40200,ASC(MID$(a$,i,1)):NEXT
```

Again, relocation notes will be given after VARCHAR.

Speaking of which, let's look at this final large character printing routine. This uses graphics operations, as an examination of the listing will show.

VARCHAR

This routine is a versatile program for printing characters out at a variety of heights. Width of the characters is constant, and has been chosen for maximum readability in all screen modes. The character printed will be located at the current graphics cursor position and will be printed in the current graphics colour.

Entry Requirements: If used with CALL, CALL address,n,m is the form. n is the ASCII code of the character to be printed and m is an integer specifying the relative height.
If called from a machine code program, A holds the value 2 and IX points to a block of memory. (IX+Ø) will hold the height, and (IX+2) will hold the ASCII code of the character.

Exit Conditions: All Registers Corrupt. Routine is exited on completion or if there is an incorrect number of parameters.

Length: 238 Bytes.

VARCHAR

```
1000  MEMORY 39999
1010  GOSUB 1090
1020  CLS
1030  INPUT a$
1040  INPUT "SIZE",A%
1050  MOVE 100,100
1060  FOR i=1 TO LEN(a$) : MOVE i*80+10,100 : CALL 40200,ASC(
      MID$(a$,i,1)),A%
1070  NEXT
1080  GOTO 1020
1090  |ASSEMBLE
1100  '        LIST
1110  '        ORG      40200
1120  '        CP       2
1130  '        RET      NZ            ; if not two parameters
1140  '                               ; return
1150  '        LD       A,(IX)
1160  '        LD       (HEIGHT),A    ; pick up height
1170  '        CALL     &B906         ; page in rom
1180  '        PUSH     AF            ; save ROM status
```

```
1190 '          LD      A,(IX+2)
1200 '                                  ; printed
1210 '          CALL    &BBA5           ; get pattern address
1220 '          LD      IX,40000        ; this is where the
1230 '                                  ; pattern is to go
1240 '          LD      B,8             ; 8 bytes to move
1250 ' LOOP     LD      A,(HL)          ; HL holds pattern
1260 '                                  ; address
1270 '          LD      (IX),A
1280 '          INC     HL
1290 '          INC     IX
1300 '          DJNZ    LOOP            ; transfer the 8 bytes
1310 '                                  ; of the pattern
1320 '          POP     AF
1330 '          CALL    &B90C           ; restore ROM state
1340 '          LD      IX,40000
1350 '          LD      D,8             ; 8 bytes of character
1360 '                                  ; definition
1370 ' LOOP1    LD      A,(IX)          ; get byte pointed
1380 '                                  ; to by IX
1390 '          LD      C,A
1400 '          LD      B,8
1410 ' LOOP2    SLA     C               ; step through each
1420 '                                  ; bit of byte
1430 '          CALL    GETCURS
1440 '          JR      C,ZERO
1450 '          CALL    VERTS
1460 '                                  ; bit is '0'
1470 '          JR      OUT
1480 ' ZERO     CALL    VERT
1490 '                                  ; bit is '1'
1500 '
1510 ' .OUT
1520 '          DJNZ    LOOP2
1530 '          INC     IX
1540 '          CALL    OUTH
1550 '          DEC     D
1560 '          JR      NZ,LOOP1        ; if all byte of
1570 '                                  ; definition have been
1580 '                                  ; done...
1590 '          RET                     ; finish
1600 ' PRINTBLOCK PUSH      HL
1610 '            PUSH      DE
1620 '            PUSH      BC
1630 '            LD        DE,6
1640 '            LD        HL,0
1650 '            CALL      &BBF9 ; draw a short
1660 '                            ; horizontal line
1670 '                            ; draw is relative
1680 '            POP       BC
1690 '            POP       DE
1700 '            POP       HL
1710 '            RET
1720 '
1730 ' PRINTSPACE PUSH      HL
1740 '            PUSH      DE
1750 '            PUSH      BC
1760 '            LD        DE,6
1770 '            LD        HL,0
1780 '            CALL      &BBC3  ; move along the
1790 '                             ; line
1800 '                             ; move is relative
1810 '            POP       BC
```

```
1820 '          POP     DE
1830 '          POP     HL
1840 '          RET
1850 ' .OUTH    PUSH    BC
1860 '          PUSH    HL
1870 '          PUSH    DE
1880 '          LD      HL,0          ; work out vertical
1890 '          LD      A,(HEIGHT)    ; relative move from

1900 '          LD      B,A           ; height parameter
1910 ' DLOOP    DEC     HL            ; so that the next
1920 '          DEC     HL            ; line of the char
1930 '          DJNZ    DLOOP         ; can be drawn
1940 '          LD      DE,-48        ; back 8*6 pixels
1950 '          CALL    &BBC3         ; relative move
1960 '          POP     DE
1970 '          POP     HL
1980 '          POP     BC
1990 '          RET
2000 '
2010 ' GETCURS  PUSH    BC
2020 '          PUSH    HL
2030 '          PUSH    DE
2040 '          PUSH    AF
2050 '          CALL    &BBC6         ; get cur graph
2060 '                                ; cursor pos....
2070 '          LD      B,6
2080 ' CLOOP    INC     DE            ; add 6 to x part
2090 '          DJNZ    CLOOP
2100 '          LD      (TEMPX),DE ; store it
2110 '          LD      (TEMPY),HL ; store the y bit
2120 '          POP     AF
2130 '          POP     DE
2140 '          POP     HL
2150 '          POP     BC
2160 '          RET
2170 ' RESTORE  PUSH    BC
2180 '          PUSH    DE
2190 '          PUSH    HL
2200 '          LD      DE,-6 ; relative move of 6
2210 '                        ; pixels to the left
2220 '          LD      HL,-2 ; and two down
2230 '          CALL    &BBC3
2240 '          POP     HL
2250 '          POP     DE
2260 '          POP     BC
2270 '          RET
2280 ' NEXT     PUSH    AF
2290 '          PUSH    BC
2300 '          PUSH    DE
2310 '          PUSH    HL
2320 '          LD      DE,(TEMPX)
2330 '          LD      HL,(TEMPY)
2340 '          CALL    &BBC0         ; absolute move to
2350 '                                ; next 'pixel' pos.
2360 '                                ; so that next part
2370 '                                ; of line of char
2380 '                                ; can be plotted
2390 '          POP     HL
2400 '          POP     DE
2410 '          POP     BC
2420 '          POP     AF
2430 '          RET
```

```
2440  ' VERT     PUSH     BC           ; draw several
2450  '          CALL     GETCURS      ; horizontal lines,
2460  '          LD       A,(HEIGHT)   ; below each other
2470  '          LD       B,A          ; to give varying
2475  '                                ; height
2480  ' VLOOP    CALL     PRINTBLOCK
2490  '          CALL     RESTORE
2500  '          DJNZ     VLOOP
2510  '          CALL     NEXT
2520  '          POP      BC
2530  '          RET
2540  ' VERTS     PUSH    BC           ; as VERT, no lines
2550  '          CALL     GETCURS      ; drawn
2560  '           LD      A,(HEIGHT)
2570  '           LD      B,A
2580  ' VLOOPS   CALL     PRINTSPACE
2590  '          CALL     RESTORE
2600  '          DJNZ     VLOOPS
2610  '          CALL     NEXT
2620  '          POP      BC
2630  '          RET
2640  '
2650  '
2660  '
2670  '
2680  ' TEMPX     WORD    0
2690  ' TEMPY     WORD    0
2700  ' HEIGHT    BYTE    0
2710  ' END
2720  RETURN
```

```
FE 02 C0 DD 7E 00 32 F5 9D CD 06 B9 F5 DD 7E 02 CD A5 BB
DD 21 40 9C 06 08 7E DD 77 00 23 DD 23 10 F7 F1 CD 0C B9
DD 21 40 9C 16 08 DD 7E 00 4F 06 08 CB 21 CD 8C 9D 38 05
CD DC 9D 18 03 CD C7 9D 10 EF DD 23 CD 74 9D 15 20 E1 C9
E5 D5 C5 11 06 00 21 00 00 CD F9 BB C1 D1 E1 C9 E5 D5 C5
11 06 00 21 00 00 CD C3 BB C1 D1 E1 C9 C5 E5 D5 21 00 00
3A F5 9D 47 2B 2B 10 FC 11 D0 FF CD C3 BB D1 E1 C1 C9 C5
E5 D5 F5 CD C6 BB 06 06 13 10 FD ED 53 F1 9D 22 F3 9D F1
D1 E1 C1 C9 C5 D5 E5 11 FA FF 21 FE FF CD C3 BB E1 D1 C1
C9 F5 C5 D5 E5 ED 5B F1 9D 2A F3 9D CD C0 BB E1 D1 C1 F1
C9 C5 CD 8C 9D 3A F5 9D 47 CD 54 9D CD A4 9D 10 F8 CD B4
9D C1 C9 C5 CD 8C 9D 3A F5 9D 47 CD 64 9D CD A4 9D 10 F8
CD B4 9D C1 C9 00 00 00 00 00
```

Notes Due to the extensive use of subroutines, this routine is not readily relocatable. However, if you are feeling adventurous, the following points should be borne in mind:

(i) The workspace address, currently at 40000, will need to be changed if the address of the program is changed so that it occupies this region.

(ii) The bytes making up the subroutine addresses in the various CALL instructions should obviously be changed.

(iii) The Assembler Directives 'WORD' and 'BYTE' are 'DEFW' and 'DEFB' on most assembler programs.

The bytes given above are for address 40200. As an example of its use, you might like to try the below BASIC program. I am assuming that the machine code is in the machine at the correct address.

```
 10  MODE 2
 20  INPUT "String ",a$
 30  INPUT "Size ",a%
 40  MOVE 100,100:REM start pos. of string
 50  FOR i=1 TO LEN(a$)
 60  MOVE i*50+10,100: REM move to next char. position
 70  CALL 40200,ASC(MID$(a$,i,1)),a%
 80  NEXT
 90  INPUT "Press ENTER to go on",a$
100  GOTO 10
```

The separation between printed characters can be adjusted by altering the MOVE statement in line 60. The character is printed with its top left hand corner at the current graphics cursor position.

Let's take a little time out from our routines to examine some of the ROM routines that have been used above. We'll also look at some limitations on where the machine code of the above three routines can be relocated to, and why.

ROM Routines Used

&B906 This enables us to switch in the lower ROM instead of the RAM that is normally in this area of the memory map. The Lower ROM occupies the memory from &0000, to &4000, and we need to have it available to us so that we can copy the Character definitions in the ROM in to the workspace at address 40000 in these programs. On leaving this routine, the A register carries what is known as the status of the ROMs, so that we can, at a later point in the program, page out the ROM and get the RAM back.

&B90C The routine at this address requires the status byte that was generated by the Operating System when we paged a ROM in, and it sets the ROM's back in to the state that they were in originally. We use this to set things back to normal after we've copied the Character definitions from ROM in to RAM.

The use of these two routines to get access to the ROM is why we have to be careful if we relocate the machine code. Think about it; if we page out RAM in which our machine code is situated, our program will crash! For this reason, we should only situate these programs in an area of memory that will NOT be affected when we page RAM out to gain access to ROM. Any address between &4000 and &BFFF will not be affected when we page ROM in; thus you can relocate your program to any address within this area, but you should not relocate the program or workspace to addresses in the range &0000 to &3FFF.

The other ROM addresses used by these routines are:

&BBA5 This routine, called by Amsoft TXT GET MATRIX allows us to get the address of the first of the 8 bytes that define the pattern that is printed to the screen when a character is printed. On calling this routine, A should hold the ASCII code. On return HL holds the address.

&BBA8 DCHAR uses this to define the upper and lower halves of the character. For our purposes, on entry HL holds the address of the first of the 8 sequential bytes that make up the definition, and the A register holds the ASCII code of the character that is to be defined.

A more detailed examination of ROM routines can be obtained in the Amstrad Firmware Manual or in "The Ins and Outs of the Amstrad".

The next two routines that we'll look at lack immediate practical use but are quite interesting all the same! Both MIRRORV and MIRRORH can be relocated anywhere in the region of memory between &4000 to &BFFF. From now on, this area of memory will be called the **Memory Pool.**

MIRRORV

This routine prints the character whose ASCII code is passed to it 'upside down' on the screen. It is as if a mirror has been put at the base of the character. The character is printed at the text cursor in the current text colour. The routine is relocatable within the memory pool.

Entry Requirements:	As the routine stands, it can be called with CALL address,n where n is the ASCII code of the character of interest. If called from another machine code program, then IX must point to the character code and A=1.
Exit Conditions:	All registers corrupt.
Length:	50 Bytes.

```
MIRRORV


1000  MEMORY 39999
1010  GOSUB 1100
1020  CLS
1030  INPUT "String : ",a$
1040  LOCATE 10,10 : PRINT A$
```

```
1050  LOCATE 10,11 : FOR I=1 TO LEN(A$) : CALL 40200,ASC(MID$
      (A$,I,1))
1060  NEXT
1070  PRINT
1080  INPUT A$ : GOTO 1020
1090  END
1100   ASSEMBLE
1110  '         LIST
1120  '         ORG       40200
1130  '         CP        1
1140  '         RET       NZ            ; if not two parameters
1150  '                                 ; return
1160  '         CALL      &B906         ; page in rom
1170  '         PUSH      AF            ; save ROM status
1180  '         LD        A,(IX)
1190  '                                 ; printed
1200  '         CALL      &BBA5         ; get pattern address
1210  '         LD        IX,40000      ; this is where the
1220  '         LD        DE,7
1230  '         ADD       HL,DE
1240  '                                 ; pattern is to go
1250  '         LD        B,8
1260  '  LOOP   LD        A,(HL)        ; HL holds address of
1270  '                                 ; last byte of pattern
1280  '         LD        (IX),A
1290  '         DEC       HL
1300  '         INC       IX
1310  '         DJNZ      LOOP          ; transfer the 8 bytes
1320  '                                 ; of the pattern so
1330  '                                 ; that it's upside down
1340  '         POP       AF
1350  '         CALL      &B90C         ; restore ROM state
1360  '         LD        HL,40000      ; redefine CHR$255
1370  '         LD        A,255
1380  '         CALL      &BBA8
1390  '         LD        A,255         ; print CHR$(255)
1400  '         CALL      &BB5A
1410  '         RET
1420  '  END
1430  RETURN
```

```
FE 01 C0 CD 06 B9 F5 DD 7E 00 CD A5 BB DD 21 40 9C 11 07
00 19 06 08 7E DD 77 00 2B DD 23 10 F7 F1 CD 0C B9 21 40
9C 3E FF CD A8 BB 3E FF CD 5A BB C9
```

Notes The routine can be relocated anywhere within the memory pool, but remember to alter the workspace address (from 40000) if you move the program in to this area of memory. To see the routine in action, try the below BASIC program.

```
100  MODE 1
110  INPUT "String ",a$
120  LOCATE 10,10:PRINT a$
130  LOCATE 10,11:
140  FOR I=1 TO LEN(a$)
150  CALL 40200,ASC(MID$(a$,I,1)):REM assumes program at
```

```
160  REM address 40200 in memory
170  NEXT
180  PRINT
190  INPUT "Press ENTER to go on",a$
200  GOTO 100
```

MIRRORH

This routine is similar to the one above but it prints a 'mirror image' of the character. The character is reflected down its right hand edge. The character is printed in the current text colour at the text cursor position.

Entry Requirements: If used with CALL, a single parameter is used. This is the ASCII code of the character to be printed.
If called from machine code, IX must point to the ASCII code and A=1.

Exit Conditions: All registers corrupt.

Length: 56 Bytes.

MIRRORH

```
1000  MEMORY 39999
1010  GOSUB 1100
1020  CLS
1030  INPUT "String :  ",a$
1040  LOCATE 10,10 : PRINT A$
1050  LOCATE 10,11 : FOR I=1 TO LEN(A$) : CALL 40200,ASC(MID$
      (A$,I,1))
1060  NEXT
1070  PRINT
1080  INPUT A$ : GOTO 1020
1090  END
1100  'ASSEMBLE
1110  '         LIST
1120  '         ORG     40200
1130  '         CP      1
1140  '         RET     NZ              ; if not two parameters
1150  '                                 ; return
1160  '         CALL    &B906           ; page in rom
1170  '         PUSH    AF              ; save ROM status
1180  '         LD      A,(IX)
1190  '                                 ; printed
1200  '         CALL    &BBA5           ; get pattern address
1210  '         LD      IX,40000        ; this is where the
1220  '                                 ; pattern is to go
1230  '         LD      B,8
1240  ' LOOP    LD      A,(HL)          ; HL holds address of
1250  '                                 ; first byte of pattern
1260  '         PUSH    BC
1270  '         LD      B,8             ; 8 bits in byte
1280  '         LD      C,A             ; get byte of def.
1290  ' ILOOP   RL      C               ; move bit from left
1300  '         RRA                     ; of C into A
1310  '         DJNZ    ILOOP
```

```
1320  '        POP     BC           ; restore BC
1330  '        LD      (IX),A       ; transfer modified
1340  '                             ; byte of def.
1350  '        INC     HL
1360  '        INC     IX
1370  '        DJNZ    LOOP         ; transfer the 8 bytes
1380  '                             ; of the pattern
1400  '        POP     AF
1410  '        CALL    &B90C        ; restore ROM state
1420  '        LD      HL,40000     ; redefine CHR$255
1430  '        LD      A,255
1440  '        CALL    &BBA8
1450  '        LD      A,255        ; print CHR$(255)
1460  '        CALL    &BB5A
1470  '        RET
1480  ' END
1490  RETURN
```

```
FE 01 C0 CD 06 B9 F5 DD 7E 00 CD A5 BB DD 21 40 9C 06 08
7E C5 06 08 4F CB 11 1F 10 FB C1 DD 77 00 23 DD 23 10 ED
F1 CD 0C B9 21 40 9C 3E FF CD A8 BB 3E FF CD 5A BB C9
```

Notes Again, the routine is relocatable within the memory pool, providing that care is taken with the workspace address. The bytes above are for address 40200.

It can be demonstrated by the BASIC routine given for MIRRORV.

We've seen enough of characters. Let's get down to some real graphics. The Amstrad OS makes using graphics operations from machine code rather easy; routines exist to plot points, drawn lines, move the graphics cursor and so on. The first graphics routine that I will describe is a sort of graphical SPRINT. It accepts a table of "instructions" and graphics coordinates and operates on them to draw the graphic shape specified by the table of data. The table accepted by this routine is called a Shape Table, 'cos it defines a shape! This is the best general approach to graphics in machine code, because each person will want to draw different shapes to the screen. Of course, for a given graphics job it might be quicker to write a routine to do that job specially, but this routine will be found to be very versatile and expandable.

GDRAW

A general purpose routine to draw graphics shapes to the screen in any screen mode. Data for the shapes to be drawn is to be found in a Shape Table, the structure of which is outlined in the notes below.

Entry Requirements: None. Program sets up all registers, but see the Notes below.

Exit Conditions: All registers are corrupt.

Length: 99 Bytes.

GDRAW

```
1000  MODE 2
1010  MEMORY 39999
1020  GOSUB 1060
1030  MODE 0
1040  CALL 40200
1050  END
1060  ASSEMBLE
1070  '          ORG      40200
1080  ' GDRAW    LD       IX,SHAPET  :  get address of data
1090  ' LOOP     LD       A,(IX)     :  first byte of 5
1100  '                              :  will be an op code
1110  '          CP       0
1120  '          RET      Z
1130  '          CP       1
1140  '          CALL     Z,MOVE     :  CALL when appropiate
1150  '          CP       2
1160  '          CALL     Z,DRAW
1170  '          CP       3
1180  '          CALL     Z,COLOUR
1190  '          CP       4
1200  '          CALL     Z,PLOT
1210  '          INC      IX         :  IX incremented to
1220  '                              :  point to next op code
1230  '          JR       LOOP
1240  ' COLOUR   PUSH     AF         :  preserve AF so that
1250  '                              :  on exit, A still has
1260  '                              :  op code in it
1270  '          INC      IX
1280  '          LD       A,(IX)     :  get colour
1290  '          CALL     &BBDE      :  change graphics pen
1300  '          INC      IX
1310  '          INC      IX
1320  '          INC      IX         :  update IX
1330  '          POP      AF
1340  '          RET
1350  ' MOVE     PUSH     AF
1360  '          CALL     COORDS
1370  '          CALL     &BBC0      :  absolute move
1380  '          POP      AF
1390  '          RET
1400  ' DRAW     PUSH     AF
1410  '          CALL     COORDS
1420  '          CALL     &BBF6      :  absolute draw
1430  '          POP      AF
1440  '          RET
1450  ' PLOT     PUSH     AF
1460  '          CALL     COORDS
1470  '          CALL     &BBEA      :  absolute plot
1480  '          POP      AF
1490  '          RET
1500  ' COORDS   INC      IX         :  routine gets the x
1510  '                              :  and y coords in
1520  '                              :  to DE and HL for
1530  '                              :  ROM routines
1540  '          LD       E,(IX)
1550  '          INC      IX
1560  '          LD       D,(IX)     :  X coordinate in DE
1570  '          INC      IX
1580  '          LD       L,(IX)
1590  '          INC      IX
```

```
1600  '          LD      H,(IX)     ; Y coordinate in HL
1610  '          RET
1620  ' SHAPET   BYTE    2          ; data from here
1630  '          WORD    200
1640  '          WORD    0
1650  '          BYTE    3
1660  '          WORD    3
1670  '          WORD    0
1680  '          BYTE    2
1690  '          WORD    200
1700  '          WORD    200
1710  '          BYTE    3
1720  '          WORD    4
1730  '          WORD    0
1740  '          BYTE    2
1750  '          WORD    0
1760  '          WORD    200
1770  '          BYTE    3
1780  '          WORD    5
1790  '          WORD    0
1800  '          BYTE    2
1810  '          WORD    0
1820  '          WORD    0
1830  '          BYTE    0    ; terminator
1840  ' END
1850  RETURN
```

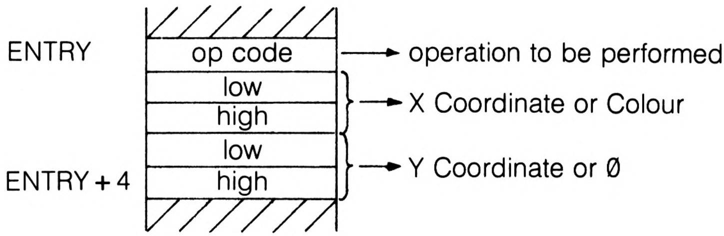
```
DD 21 6B 9D DD 7E 00 FE 00 C8 FE 01 CC 3B 9D FE 02 CC 44
9D FE 03 CC 2A 9D FE 04 CC 4D 9D DD 23 18 E2 F5 DD 23 DD
7E 00 CD DE BB DD 23 DD 23 DD 23 F1 C9 F5 CD 56 9D CD C0
BB F1 C9 F5 CD 56 9D CD F6 BB F1 C9 F5 CD 56 9D CD EA BB
F1 C9 DD 23 DD 5E 00 DD 23 DD 56 00 DD 23 DD 6E 00 DD 23
DD 66 00 C9 02 C8 00 00 00 03 03 00 00 00 02 C8 00 C8 00
03 04 00 00 00 02 00 00 C8 00 03 05 00 00 00 02 00 00 00
00 00
```

Notes The address of the Shape Table is loaded into the IX register early on in the program. You might like to alter this so that when the program is called, the user can supply a Shape Table address to the program. The routine is relocatable within the memory Pool provided that, of course, the subroutine call addresses (Not the OS Routine Call Addresses!) are modified to take the new addresss into account.

When the routine starts acting upon the graphics instructions stored in the Shape Table, it will start operations in the current graphics colour and at the current graphics cursor. Thus, if there is any doubt about these when the routine is entered, you should ensure that the first couple of entries in the Shape Table set the colour and cursor appropriately.

The Shape Table

This consists of a series of 5 byte entries, all in the below format.



The 5 byte entry is repeated for each operation to be performed by GDRAW. The op codes understood by this routine are shown in the Table below.

Table of GDRAW Op Codes

Op Code	Operation
Ø	Finish
1	Absolute Move
2	Absolute Draw
3	Colour
4	Absolute Plot

Any sequence of operations should thus end with an all zero entry, to indicate the fact. For the Move, Draw and Plot operations the X and Y coordinates are stored in two bytes with the Low byte first. For Colour, which specifies which of the colours available will be used next, the low byte of the X coordinate entry is used to hold the colour code, which will be the number of the graphics pen that you want to use. All other entries for Colour will be zero.

It is easy to add more op codes to the routine, and if you want to do this, to include, for example, relative Move and Draw instructions, then the following will be helpful.

(i) The AF register should be preserved on entry to the routine that services the new Op Code. This ensures that the A register can be restored to its original value on leaving the routine. This is important, as there is otherwise the chance for the program to call a second subroutine before updating the IX register and getting a new Op Code.

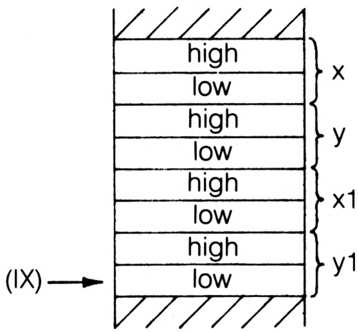
(ii) It's probably a good idea to leave Ø as the finish Op Code, unless you want to alter the program.

I'll now give you some routines that are dedicated to producing certain shapes, such as triangles and rectangles.

TDRAW

This routine draws a triangle whose angles are at the last point visited by the graphics cursor and the two points passed to the machine code routine as parameters. The triangle is drawn in the current graphics ink.

Entry Requirements: If called with a CALL statement, CALL address,x,y,x1,y1 is the correct form, x,y and x1,y1 being the two angles of the triangle. If called from another machine code program, then IX points to a parameter block such as that on the left. A =4.



Exit Conditions: All Registers Corrupt.

Length: 44 Bytes.

TDRAW

```
1000  MODE 1
1010  REM Triangle drawing routine
1020  GOSUB 1080
1030  CLS
1040  INPUT "First point",x,y
1050  INPUT "Second point",x1,y1
1060  CALL 40200,x,y,x1,y1
1070  GOTO 1040
1080  |ASSEMBLE
1090  '            org       40200
1100  '            CP        4          ; are there 4 param
1110  '            RET       NZ
1120  '            CALL      &BBC6      ; get cursor position
```

```
1130  '              PUSH    DE        ; save x
1140  '              PUSH    HL        ; save y
1150  '              CALL    DRAWV     ; draw to x1,y1
1160  '              INC     IX        ; update IX point to
1170  '              INC     IX        ; x and y parameters
1180  '              INC     IX
1190  '              INC     IX
1200  '              CALL    DRAWV     ; draw to x,y
1210  '              POP     HL        ; recover original x
1220  '              POP     DE        ; and y
1230  '              CALL    &BBF6     ; draw back there
1240  '              RET
1250  ' DRAWV        LD      L,(IX)
1260  '              LD      H,(IX+1)
1270  '              LD      E,(IX+2)
1280  '              LD      D,(IX+3)
1290  '              CALL    &BBF6
1300  '              RET
1310  ' END
1320  RETURN
```

```
FE 04 C0 CD C6 BB D5 E5 CD 24 9D DD 23 DD 23 DD 23 DD 23
CD 24 9D E1 D1 CD F6 BB C9 DD 5E 00 DD 56 01 DD 6E 02 DD
66 03 CD F6 BB C9
```

Notes The program can be relocated. The bytes given above are suitable for all memory locations in the Memory Pool. When the triangle has been drawn, the graphics cursor will be at the position it was before the CALL was made.

BDRAW

This routine draws square or rectangular shapes on the screen, in either outline or filled in. Drawing and filling is in the current graphics ink. This routine is relocatable provided care is taken with the subroutine addresses used.

Entry Requirements: For the CALL statement, CALL address,xy, length,height,(n) n is optional, and if present causes a filled in box to be drawn. It's value isn't important. See Figure 1 for the other parameters.
If being called from another machine code routine, IX should point to the parameter blocks shown in Figure 2.

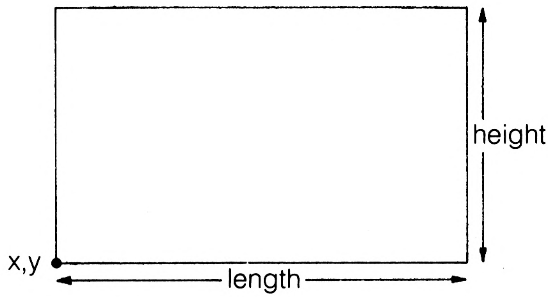


Figure 1

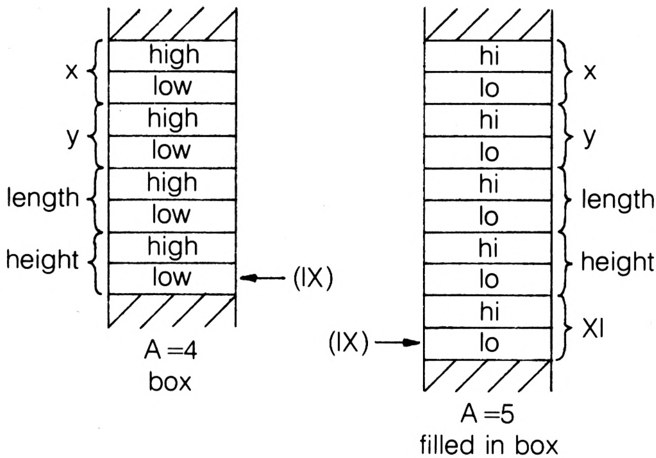


Figure 2

Exit Conditions: All Registers Corrupt.

Length: 149 Bytes.

BDRAW

```
1000  REM box drawing and filling
1010  GOSUB 1050
1020  CALL 40200,100,100,200,100
1030  CALL 40200,100,100,100,100,1
1040  END
1050   ASSEMBLE
1060                  ORG        40200
```

```
1070 '          CP      4
1080 '          JR      Z,OPENB
1090 '          CP      5          ; If 5 param, filled
1095 '                             ; filled box
1100 '          JR      Z,CLOSEB
1110 '          RET
1120 ' OPENB    CALL    MOVEP
1130 '          LD      E,(IX+2)
1140 '          LD      D,(IX+3)
1150 '          LD      HL,0
1160 '          CALL    &BBF9      ; draw bottom
1170 '          LD      DE,0
1180 '          LD      L,(IX)
1190 '          LD      H,(IX+1)
1200 '          CALL    &BBF9      ; draw right edge
1210 '          CALL    MOVEP
1220 '          LD      DE,0
1230 '          LD      L,(IX)
1240 '          LD      H,(IX+1)
1250 '          CALL    &BBF9      ; draw left edge
1260 '          LD      HL,0
1270 '          LD      E,(IX+2)
1280 '          LD      D,(IX+3)
1290 '          CALL    &BBF9      ; draw the top edge
1300 '          RET
1310 ' MOVEP    LD      L,(IX+4)
1320 '          LD      H,(IX+5)
1330 '          LD      E,(IX+6)
1340 '          LD      D,(IX+7)
1350 '          CALL    &BBC0      ; move start pos
1360 '          RET                ; of box.
1370 ' CLOSEB   CALL    &BBCC
1380 '          PUSH    DE
1390 '          PUSH    HL         ; get and save curr
1400 '                             ; graphics origin
1410 '          LD      L,(IX+6)
1420 '          LD      H,(IX+7)
1430 '          LD      E,(IX+8)
1440 '          LD      D,(IX+9)
1450 '          CALL    &BBC9      ; move grap org
1460 '                             ; to pos. of box
1470 '          LD      L,(IX+2)
1480 '          LD      H,(IX+3)   ; get number of
1490 '          PUSH    HL         ; needed to fill box
1500 '          POP     BC         ; in to BC register
1510 '          LD      HL,0
1520 ' LOOP     LD      E,(IX+4)   ; draw a series of
1530 '          LD      D,(IX+5)   ; lines of length
1540 '          PUSH    BC         ; len to fill the
1550 '          PUSH    HL         ; box up
1560 '          LD      HL,0
1570 '          CALL    &BBF9
1580 '          POP     HL
1590 '          INC     HL
1600 '          PUSH    HL
1610 '          LD      DE,00
1620 '          CALL    &BBC0
1630 '          POP     HL
1640 '          POP     BC
1650 '          DEC     BC
1660 '          LD      A,C
1670 '          OR      B
1680 '          CP      0
```

```
1690  '                    JR      NZ,LOOP  ; if not done round
1700  '                    POP     HL       ; again else restore
1710  '                    POP     DE       ; graphics origin
1720  '                    CALL    &BBC9
1730  '                    RET
1740  ' END
1750 RETURN
```

```
FE 04 28 05 FE 05 28 48 C9 CD 48 9D DD 5E 02 DD 56 03 21
00 00 CD F9 BB 11 00 00 DD 6E 00 DD 66 01 CD F9 BB CD 48
9D 11 00 00 DD 6E 00 DD 66 01 CD F9 BB 21 00 00 DD 5E 02
DD 56 03 CD F9 BB C9 DD 6E 04 DD 66 05 DD 5E 06 DD 56 07
CD C0 BB C9 CD CC BB D5 E5 DD 6E 06 DD 66 07 DD 5E 08 DD
56 09 CD C9 BB DD 6E 02 DD 66 03 E5 C1 21 00 00 DD 5E 04
DD 56 05 C5 E5 21 00 00 CD F9 BB E1 23 E5 11 00 00 CD C0
BB E1 C1 0B 79 B0 FE 00 20 E0 E1 D1 CD C9 BB C9
```

Notes There are several interesting points to note about this routine. The first is the use of the number of parameters passed to the routine to specify which of the two options, 'filled' or 'open' boxes, are drawn. This is easily done by virtue of the fact that on entry to a machine code routine using CALL the number of parameters in the CALL statement is passed over in the A register. The value of parameter 'n' is of no importance; it is its presence that causes the routine to be entered.

The other interesting points about this routine are the ROM routines used and the method used to fill the boxes when necessary.

&BBCC This allows us to get the current position of the origin used for the graphics operations. That is, the point used as Ø,Ø in all graphics operations. On exit, HL contains the y coordinate and DE contains the x coordinate. We need this information because we alter the origin when we draw the filled in boxes, and it's nice to restore things back to normal before we go back to BASIC.

&BBC9 This ROM routine entry point allows us to set the graphics origin to a particular x,y coordinate.. DE holds the x coordinate and HL holds the y coordinate. A call is then made to the routine. It's used in this program to restore the graphics origin to it's original position before leaving the routine.

Although there is a routine within the Amstrad ROM to fill an area of the screen with colour, it requires the pixel coordinates to be converted into screen addresses; I decided to take the easy way out, and simply draw lines to fill the area of the box. This isn't as fast as the resident fill routine, but is simpler to set up and use.

As a possible extension to this program, you could allow the 'n' parameter to specify the colour that a filled box is to be drawn in. The colour could be set using the GRA SET PEN routine that we discussed briefly in Chapter 2.

Circle Drawing

A very common requirement in programming is to be able to draw circles quickly. There are problems with using machine code to do this however; 'real' numbers are involved in the calculation of the Sines and Cosines used in circle drawing routines, and so this would mean that the machine code routines written by us to calculate these values would be rather long and not much, if at all, faster than BASIC. So, here I present a couple of fairly useful BASIC routines for general purpose circle drawing, and a hybrid machine code-BASIC program for special cases.

CIRCLE1

This is a straight forward routine for drawing circles or most polygons. This makes it rather useful. Any value of 'n' in the program will give a shape of some sort, with the exception of Ø, 1 and 2. Note that values of n=5, n=1Ø or n=5Ø give a gap in the shape that is drawn. Values of 'n' greater than 25 give a reasonable circle.

```
CIRCLE1

10      REM circle 1
20      REM Polygon drawer. n values above about
30      REM 25 give a circle. Higher the value
40      REM of n, the better the circle but it
50      REM takes longer to draw
60      MODE 2
70      INPUT "Number of sides  ",n
80      xc=200 : yc=200 : REM centre of circle
90      cstep=6.28/n : REM increment needed
100     cend=6.28
110     crad=100 : REM radius of shape
120     MOVE xc+crad,yc
130     FOR i=0 TO cend STEP cstep
140     DRAW xc+crad*COS(i),yc+crad*SIN(i) : REM do it
150     NEXT
160     END
```

CIRCLE2

This routine is faster than the last but is not as versatile in that it cannot draw other polygons. Indeed, the CIRCLE1 program is worthy of a little experiment. The extra speed in this program is by virtue of the fact that the time consuming job of calculating Sines and Cosines is only done once. The other sine and cosine values that are required are calculated from these initial values.

CIRCLE2

```
10     MODE 2
20     REM circle 2
30     REM faster circles draw here
40     REM though no polygons.
50     crad=100 : REM radius of circle
60     cx=100 : cy=100 : REM centre of circle
70     c=COS(3.14/25) : s=SIN(3.14/25)
80     oldc=1 : oldsin=0
90     MOVE cx+crad*oldc,cy+crad*oldsin
100    FOR i=1 TO 50
110    newc=oldc*c-oldsin*s
120    newsin=oldsin*c+oldc*s
130    DRAW cx+crad*newc,cy+crad*newsin
140    oldc=newc : oldsin=newsin
150    NEXT
```

CIRCLE3

We now come to the hybrid method of drawing circles, which effectively uses a variation on the GDRAW program that we saw at the start of this Chapter. A BASIC routine works out the coordinates for the particular circle that is to be drawn, and stores the values thus obtained in a table where the machine code program can access the coordinates and draw them to the screen when required. The disadvantage in this method is that it is useful for only a particular radius circle at a particular position on the screen; to draw a different circle it is necessary to redefine the values in the data table used by the program by re-running the BASIC part of the program with new parameters.

CIRCLE3

```
1000   MODE 2
1010   REM circle 3
1020   REM faster circles draw here
1030   REM circle is fixed position and
1040   REM fixed radius
1050   GOSUB 1270
1060   crad=100 : REM radius of circle
1070   cx=100 : cy=100 : REM centre of circle
1080   c=COS(3.14/25) : s=SIN(3.14/25)
1090   oldc=1 : oldsin=0
1100   coords=40300
1110   xcoord=cx+crad*oldc : GOSUB 1230
1120   xcoord=cy+crad*oldsin : GOSUB 1230
1130   FOR i=1 TO 50
1140   newc=oldc*c-oldsin*s
1150   newsin=oldsin*c+oldc*s
1160   xcoord=cx+crad*newc : GOSUB 1230
1170   xcoord=cy+crad*newsin : GOSUB 1230
1180   oldc=newc : oldsin=newsin
1190   NEXT
1200   CLS : INPUT "Press Enter to draw Circle",a$
1210   CALL 40200
```

```
1220  END
1230  a$=HEX$(xcoord) : a$=RIGHT$("0000"+a$,4)
1240  lo=VAL("&"+RIGHT$(a$,2)) : hi=VAL("&"+LEFT$(a$,2))
1250  POKE coords,lo : coords=coords+1 : POKE coords,hi : coo
      rds=coords+1
1260  RETURN
1270  REM assembles machine code
1280   |ASSEMBLE
1290  '               org      40200
1300  '               LD       IX,40300
1310  '               LD       E,(IX)
1320  '               LD       D,(IX+1)
1330  '               LD       L,(IX+2)
1340  '               LD       H,(IX+3)
1350  '               CALL     &BBC0
1360  '               INC      IX
1370  '               INC      IX
1380  '               INC      IX
1390  '               INC      IX
1400  '               LD       B,50
1410  ' LOOP          LD       E,(IX)
1420  '               INC      IX
1430  '               LD       D,(IX)
1440  '               INC      IX
1450  '               LD       L,(IX)
1460  '               INC      IX
1470  '               LD       H,(IX)
1480  '               INC      IX
1490  '               PUSH     BC
1500  '               CALL     &BBF6
1510  '               POP      BC
1520  '               DJNZ     LOOP
1530  '               RET
1540  ' END
1550  RETURN
```

```
DD 21 6C 9D DD 5E 00 DD 56 01 DD 6E 02 DD 66 03 CD C0 BB
DD 23 DD 23 DD 23 DD 23 06 32 DD 5E 00 DD 23 DD 56 00 DD
23 DD 6E 00 DD 23 DD 66 00 DD 23 C5 CD F6 BB C1 10 E5 C9
```

The Assembler listing given above requires the coordinates for the circle to be stored in memory starting at address 4Ø3ØØ. The circle drawing routine gets the first x and y coordinate and uses them to do a MOVE operation to a point on the radius of the circle. Subsequent points draw the rest of parameter of the circle. The below program will load in the coordinate information, and call the routine to draw the circle, which is expected to be at address 4Ø3ØØ, with the data at address 4Ø2ØØ.

```
10  crad=100:REM radius of circle
20  cx=100:cy100:REM centre of circle
30  c=COS(3.14/25):s=SIN(3.14/25)
40  oldc=1:oldsin=0
50  coords=40300: REM address of table
```

```
 6Ø  xcoord =cx+crad*oldc: GOSUB 8ØØ
 7Ø  xcoord =cy+crad*oldsin:GOSUB 8ØØ
 8Ø  FOR I=1 TO 5Ø
 90  newc =oldc*c-oldsin*s
1ØØ  newsin =oldsin*c +oldc*s
11Ø  xcoord =cx+crad*newc:GOSUB 8ØØ
12Ø  ycoord =cy+crad*newsin:GOSUB 8ØØ
13Ø  oldc =newc:oldsin =newsin
14Ø  NEXT
15Ø  CLS:INPUT "Press Enter to draw Circle",a$
16Ø  CALL 4Ø2ØØ: REM assuming the routine
17Ø  REM is at this address
18Ø  END
8ØØ  REM Subroutine to be discussed in the
81Ø  REM Notes below
82Ø  a$=HEX$(xcoord):a$=RIGHT$("ØØØØ"+a$,4)
83Ø  lo=VAL("&"+RIGHT$(a$,2))
84Ø  hi=VAL("&"+LEFT$(a$,2))
85Ø  POKE coords,lo
86Ø  coords =coords+1
87Ø  POKE coords,hi
88Ø  coords =coords+1
89Ø  RETURN
```

The subroutine at line 8ØØ is quite useful for storing decimal numbers in memory in the "low byte first" format that the Z8Ø expects to find all its data in. Here it is used to store the coordinate information in memory so that it can be used by the machine code drawing routine. You can probably see from the listing exactly how the routine works; it converts the number in to a string of characters representing the hexadecimal of the number, and then uses RIGHT$ and LEFT$ to extract the low and high bytes of the number.

That completes this Chapter of graphic routines, However, many of the routines to be mentioned in the next two Chapters are graphically oriented, so don't panic if you haven't found exactly what you want yet!

4. Scrolling the Screen

First of all, what is a scroll? Well, it's the process of moving a whole screen, or part of a screen, at once, retaining the displayed image. This enables us to do some rather interesting things; the screen can be moved up and down, or from side to side, or just a single word or graphics shape can be made to move across the screen. Routines exist in the firmware to do some simple vertical and horizontal scrolls, and in this Chapter we'll see some routines that use these, and other scrolling routines that work by directly accessing the video memory.

There are two main types of scroll; the software scroll, in which the scroll is carried out without the hardware that generates the screen being involved, and the hardware scroll in which the hardware responsible for generating the screen image is manipulated in some way.

We'll start with a very simple routine which moves the whole screen up and down.

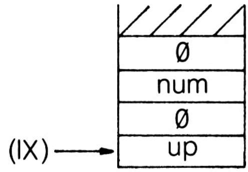
HSCROLL

A routine to scroll the whole screen up and down by a given number of character lines in any screen mode. The routine is relocatable, and lines of text that are scrolled off the top or bottom of the screen are lost forever. Lines that are 'scrolled in' to replace these lost lines are filled in the current text paper colour.

Entry Requirements: Called from BASIC with CALL address,num,up where num is the number of lines that you want scrolling and up is either Ø or 1. up=1 will scroll the screen up and up=Ø will scroll the screen down.
If called from machine code, A=2 and IX points to a parameter block. num should be in the range Ø to 255 only, although large values will simply clear the screen.

Exit Conditions: All Registers Corrupt.

Length: 22 Bytes.



HSCROLL Parameter Block

HSCROLL

```
10     MEMORY 39999
20     GOSUB 1000
30     MODE 2
40     PRINT "▲▲▲▲▲▲▲▲▲Hello"
50     PRINT "▲▲▲▲▲▲▲▲▲There"
60     PRINT "▲▲▲▲▲▲▲▲▲You"
70     PRINT "▲▲▲▲▲▲▲▲▲People!!"
80     CALL 40200,1,dir
90     a$=INKEY$ : IF a$="" GOTO 90
91     a=ASC(a$)
92     IF a=241  THEN dir=0
93     IF a=240 THEN dir=1
100    GOTO 80
1000    ASSEMBLE
1010   '          org     40200
1020   '          CALL    &BB99    : get text paper colour
1030   '          CALL    &BC2C    : encode the colour for later

1040   '          LD      B,(IX+2): number of scrolls
1050   'LOOP      PUSH    AF       : preserve the registers
1060   '          PUSH    BC
1070   '          LD      B,(IX+0): is it up or down?
1080   '          CALL    &BC4D    : do the scroll
1090   '          POP     BC
1100   '          POP     AF
1110   '          DJNZ    LOOP     : repeat until all done
1120   '          RET
1130   ' END
1140   RETURN
```

```
CD 99 BB CD 2C BC DD 46 02 F5 C5 DD 46 00 CD 4D BC C1 F1
10 F4 C9
```

Notes This makes use of a very useful firmware routine that is called at &BC4D to scroll the whole screen. On entry, if B = 0 then the screen is scrolled down one line, and if B=1 then it is scrolled up a line. The below BASIC program demonstrates the above machine code.

```
10  MODE 2
20  PRINT:PRINT:PRINT
30  PRINT "Hello There"
40  PRINT "You guys!!"
50  CALL 40200,1,dir:REM assume code at 40200
60  a$=INKEY$:IF a$="" THE GOTO 60
70  IF ASC(a$)=241 THEN dir=0:REM down
80  IF ASC(a$)=240 THEN dir=1:REM up
90  GOTO 50
```

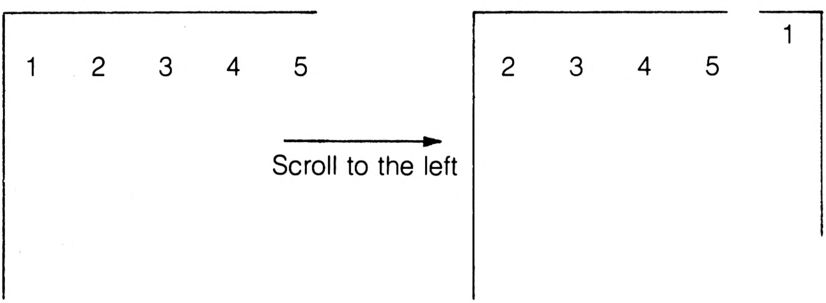
Pressing the up and down arrow keys will move the screen. The text, if scrolled off the screen is lost for good.

It would be rather useful to be able to scroll the screen sideways. This requires a little more work, as we must produce a slightly different routine for each screen mode. Before we examine these routines in detail, a few general notes.

Sideways Scrolling

These routines all scroll MOST of the screen, not all of it. Lines 0 and 24 of the display are used as 'workspace' by the routines, and so shouldn't be used if you want to use these routines. The reason for this will be made clear shortly.

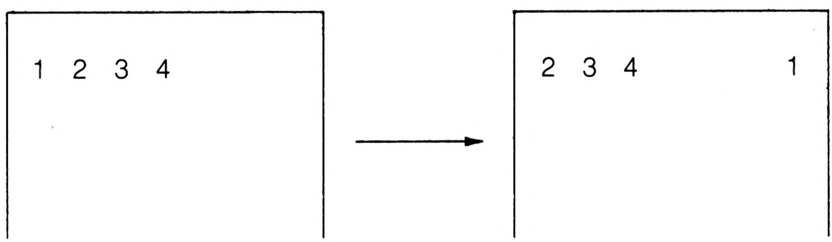
We use a mixture of hardware and software scrolls to get the effect that we want. It is easy to move sideways using hardware by altering what is known as the Screen Offset. There isn't room here to go into it in detail, and so you're directed to the Firmware manual or some similar work for full details.



You can see above how a pure sideways hardware scroll affects the display. The first character in the line moves to the opposite edge of the screen, and the other characters move to the left by a given amount. Due to the hardware of the Amstrad, this value depends on the Screen Offset in the following way.

Incrementing the screen offset leads to a scroll to the right and decrementing the Screen Offset leads to a scroll to the left. The current value of the offset can be obtained by the use of a firmware routine, as we'll soon see. A second routine can be used to write the modified screen offset back to the Video Circuitry. In these three programs, the Mode 1 and 2 offset is modified by 2 each time. In mode 1, this leads to a single character move. In mode 2, this is a 2 character scroll. Thus to scroll the mode 1 screen 1 character to the left, you simply reduce the offset by 2. In mode Ø, the screen offset is modified by 4 each time a scroll is required. This leads to a single character scroll. The reason for differing offsets required in each mode to achieve a single character scroll is due to the different lay out of Video RAM in each screen mode.

However, the observant amongst you will have noticed that the scroll shown above wasn't a true sideways scroll; the character scrolled in to the screen was a line higher or lower than the line from which it originally came, depending upon the direction of the scroll. For example, a right scroll using this method would lead to a character that is scrolled off of the right edge of the screen re-appearing on the left edge of the screen, 1 line below its start line. For a true sideways scroll, we must ensure that the material scrolled out of one side of the screen is scrolled in to the other side of the screen on the same line, in the below fashion.



This can be done using the SW SCROLL routine that is present in the Amstrad Firmware. This allows us to scroll a particular area of the screen up or down by one character row. Before looking at the scroll routine, let's briefly examine the Firmware routines that we'll use.

&BCØB This routine returns the current value of the Screen Offset in the HL register pair. This can then be modified and sent back to the Video Circuitry.

&BCØ5 This routine allows us to set the Screen Offset to a value of our choosing. On entry, the HL register pair should hold the desired value of the Offset.

&BC5Ø This routine, called SW SCROLL, allows us to vertically scroll a given area of the screen. On entry, B=Ø for a down scroll or 1 for an up scroll. H holds the left edge of the area to be scrolled, L the top row, D the right edge and E the bottom row. All these are in terms of character spaces, starting at Ø,Ø in the top left corner of the screen. The A register holds the encoded ink that is to be used to fill the line that is scrolled in. In our routines, we'll be using the text paper colour to fill the line scrolled in.

&BB99 This routine returns in the A register the current text paper colour. Before we can use the value so returned in the SW SCROLL routine, we have to encode it by a call to the routine at &BC2C.

An examination of the listings for the three sideways scroll routines will show that we use the SW SCROLL routine to 'line up' the columns of the screen display that have been scrolled off of one side of the screen and on to the opposite edge of the screen.

SSCROLL2

This scrolls a mode 2 screen sideways by 2 character spaces. Screen rows Ø and 24 are not scrolled properly. The routine can be relocated provided that the address of the workspace is altered so as not to clash with the program. The bytes given are for address 4Ø2ØØ.

Entry Requirements:	From BASIC: CALL address,dir where dir=1 for a left scroll or dir=Ø for a right scroll. From Machine code, A=1 and IX points to a single byte holding 'dir'.
Exit Conditions:	All registers corrupt.
Length:	90 Bytes.

SSCROLL2

```
1000  REM Mode 2 left and right scroll
1010  MEMORY 39999
1020  GOSUB 1130
1030  MODE 2
1040  ORIGIN 0,32 : REM set graphics origin within window scr
      olled
1050  WINDOW 1,80,2,24 : REM set up the text window to be scr
      olled
1060  MOVE 0,10 : DRAW 100,100 : DRAW 150,50 : DRAW 200,50 :
      DRAW 300,60 : DRAW 400,100 : DRAW 450,20 : DRAW 500,10 :
       DRAW 600,150 : DRAW 640,10
1070   CALL &BD19 : REM only update when screen redrawn
1080  A$=INKEY$ : IF A$="" THEN GOTO 1080
1090  IF ASC(A$)=243 THEN CALL 40200,0 : REM  detect arrow ke
      ys
1100  IF ASC(A$)=242 THEN CALL 40200,1
1110  GOTO 1070
```

```
1120  END
1130   ASSEMBLE
1140  '              ORG     40200
1150  '              DI                ; turn off interrupts
1160  '              CALL    &BB99
1170  '              CALL    &BC2C
1180  '              LD      (PAPER),A  ; save curr text pap
1190  '              LD      A,(IX)     ; decide left/right
1200  '              CP      0
1210  '              JR      Z,OTHER
1220  '             LD      H,78
1230  '             LD      L,0
1240  '             LD      D,79
1250  '             LD      E,24
1260  '              LD      B,0
1270  '              PUSH    BC        ; set up for scroll
1280  '              PUSH    HL        ; preserve reg's
1290  '              PUSH    DE        ; on the stack
1300  '              CALL    &BC0B     ; get offset
1310  '              INC     HL
1320  '              INC     HL        ; update it
1340  '              CALL    &BC05     ; and offset to 6845
1350  '              LD      A,(PAPER)
1360  '              POP     DE
1370  '              POP     HL
1380  '              POP     BC
1390  '              CALL    &BC50     ; scroll a col get
1400  '                                ; lined up
1410  '              EI                ; enable int
1420  '              RET
1430  ' OTHER        LD      H,0       ; set reg's for
1440  '              LD      L,0       ; col scroll at end
1450  '              LD      D,1       ; of routine
1460  '              LD      E,24
1470  '              LD      B,1
1480  '              LD      A,(PAPER)
1490  '              PUSH    AF
1500  '              PUSH    BC
1510  '              PUSH    DE
1520  '              PUSH    HL
1530  '              CALL    &BC0B    ; get offset
1535  '              DEC     HL
1540  '              DEC     HL       ; alter the offset
1570  '              CALL    &BC05    ; send it to 6845
1580  '              POP     HL
1590  '              POP     DE
1600  '              POP     BC
1610  '              POP     AF
1620  '              CALL    &BC50    ; scroll left col get
1630  '                               ; lined up
1640  '              EI
1650  '              RET
1660  '
1670  '
1680  '
1690  ' PAPER        BYTE    0
1700  ' END
1710  RETURN
```

```
F3 CD 99 BB CD 2C BC 32 5B 9D DD 7E 00 FE 00 28 20 26 4E
2E 00 16 4F 1E 18 06 00 C5 E5 D5 CD 0B BC 23 23 CD 05 BC
3A 5B 9D D1 E1 C1 CD 50 BC FB C9 26 00 2E 00 16 01 1E 18
06 01 3A 5B 9D F5 C5 D5 E5 CD 0B BC 2B 2B CD 05 BC E1 D1
C1 F1 CD 50 BC FB C9 00
```

Notes The very top and very bottom lines of the display are not scrolled properly, this being a by product of the way in which altering the offset affects material scrolled off of the side of the screen. The central 23 lines of the display are scrolled properly, however.

Interrupts are disabled in this routine, to attempt to get a little more speed. Also in order to provide a little more speed, note how the registers for the call to SW SCROLL have been set up early on in the routine. This cuts down the number of instructions that need to be executed between the altering of the Screen Offset and the call to SW SCROLL to clear the edge of the screen up.

SSCROLL1 and SSCROLLØ are similar routines, but are designed for use in modes 1 and Ø respectively. The below BASIC routine can be used to demonstrate all these routines in action.

```
100 MODE 2
110 ORIGIN 0,32:REM Don't use bottom screen line
120 WINDOW 1,80,2,24: REM Don't use bottom or top
130 REM lines of the text screen
140 MOVE 0,10:DRAW 100,100:DRAW 150,50:DRAW 200,50
150 DRAW 300,60:DRAW 400,100:DRAW 450,20
160 DRAW 500,10:DRAW 600,150:DRAW 640,10
170 CALL &BD19 : REM wait for next frame
180 A$=INKEY$:IF A$="" THEN GOTO 180
190 IF ASC(A$)=243 THEN CALL 40200,0
200 IF ASC(A$)=242 THEN CALL 40200,1
210 GOTO 170
```

Pressing the sideways arrow keys will cause the screen picture to scroll accordingly.

SSCROLLØ

This routine does a sideways scroll by 1 character of the mode Ø screen.

Entry Requirements: As for SSCROLL2

Exit Conditions: As for SSCROLL2

Length: 96 Bytes.

SSCROLL0

```
1000  REM Mode 0 left and right scroll
1010  MEMORY 39999
1020  GOSUB 1130
1030  MODE 0
1040  ORIGIN 0,32 : REM set graphics origin within window scr
      olled
1050  WINDOW 1,20,2,24 : REM set up the text window to be scr
      olled
1060  MOVE 0,10 : DRAW 100,100 : DRAW 150,50 : DRAW 200,50 :
      DRAW 300,60 : DRAW 400,100 : DRAW 450,20 : DRAW 500,10 :
       DRAW 600,150 : DRAW 640,10
1070   CALL &BD19 : REM only update when screen redrawn
1080  A$=INKEY$ : IF A$="" THEN GOTO 1080
1090  IF ASC(A$)=243 THEN CALL 40200,0 : REM  detect arrow ke
      ys
1100  IF ASC(A$)=242 THEN CALL 40200,1
1110  GOTO 1070
1120  END
1130   ASSEMBLE
1140  '            ORG     40200
1150  '            DI                ; disable int
1160  '            CALL    &BB99
1170  '            CALL    &BC2C
1180  '            LD      (PAPER),A ; save text paper
1190  '            LD      A,(IX)    ; left or right
1200  '            CP      0
1210  '            JR      Z,OTHER
1220  '           LD       H,19
1230  '           LD       L,0
1240  '           LD       D,19
1250  '           LD       E,24
1260  '            LD      B,0
1270  '            PUSH    BC        ; set up scroll
1280  '            PUSH    HL        ; preserve reg's
1290  '            PUSH    DE        ; on the stack
1300  '            CALL    &BC0B     ; get offset
1310  '            LD     DE,4
1320  '            ADD    HL,DE      ; update it
1330  '            LD      A,(PAPER); get paper
1340  '            CALL    &BC05     ; update offset to 6845
1350  '            LD      A,(PAPER)
1360  '            POP     DE
1370  '            POP     HL
1380  '            POP     BC
1390  '            CALL    &BC50     ; scroll col get
1400  '                              ; lined up
1410  '            EI                ; enable int
1420  '            RET
1430  ' OTHER      LD      H,0       ; set up reg's
1440  '            LD      L,0       ; col scroll at end
1450  '            LD      D,0       ; of routine
1460  '            LD      E,24
1470  '            LD      B,1
1480  '            LD      A,(PAPER)
1490  '            PUSH    AF
1500  '            PUSH    BC
1510  '            PUSH    DE
1520  '            PUSH    HL
1530  '            CALL    &BC0B    ; get offset
1540  '            XOR     A
```

```
1545   '              LD      DE,4     ; alter it and
1550   '              SBC     HL,DE
1560   '              LD      A,(PAPER)
1570   '              CALL    &BC05    ; send to 6845
1580   '              POP     HL
1590   '              POP     DE
1600   '              POP     BC
1610   '              POP     AF
1620   '              CALL    &BC50    ; scroll left col get
1630   '                               ; lined up
1640   '              EI
1650   '              RET
1660   '
1670   '
1680   '
1690   ' PAPER        BYTE    0
1700   ' END
1710   RETURN
```

```
F3 CD 99 BB CD 2C BC 32 67 9D DD 7E 00 FE 00 28 25 26 13
2E 00 16 13 1E 18 06 00 C5 E5 D5 CD 0B BC 11 04 00 19 3A
67 9D CD 05 BC 3A 67 9D D1 E1 C1 CD 50 BC FB C9 26 00 2E
00 16 00 1E 18 06 01 3A 67 9D F5 C5 D5 E5 CD 0B BC AF 11
04 00 ED 52 3A 67 9D CD 05 BC E1 D1 C1 F1 CD 50 BC FB C9
00
```

SSCROLL1

This routine performs a sideways scroll of 1 character space in Mode 1.

Entry Requirements: As for SSCROLL2

Exit Conditions: As for SSCROLL2

Length: 96 Bytes.

SSCROLL1

```
1000  REM Mode 1 left and right scroll
1010  MEMORY 39999
1020  GOSUB 1130
1030  MODE 1
1040  ORIGIN 0,32 : REM set graphics origin within window scr
      olled
1050  WINDOW 1,40,2,24 : REM set up the text window to be scr
      olled
1060  MOVE 0,10 : DRAW 100,100 : DRAW 150,50 : DRAW 200,50 :
      DRAW 300,60 : DRAW 400,100 : DRAW 450,20 : DRAW 500,10 :
       DRAW 600,150 : DRAW 640,10
1070   CALL &BD19 : REM only update when screen redrawn
1080  A$=INKEY$ : IF A$="" THEN GOTO 1080
1090  IF ASC(A$)=243 THEN CALL 40200,0 : REM  detect arrow ke
      ys
1100  IF ASC(A$)=242 THEN CALL 40200,1
1110  GOTO 1070
1120  END
```

```
1130  ASSEMBLE
1140  '              ORG    40200
1150  '              DI                   ; disable int
1160  '              CALL   &BB99
1170  '              CALL   &BC2C
1180  '              LD     (PAPER),A ; save text paper
1190  '              LD     A,(IX)    ; left or right
1200  '              CP     0
1210  '              JR     Z,OTHER
1220  '             LD      H,39
1230  '             LD      L,0
1240  '             LD      D,39
1250  '             LD      E,24
1260  '              LD     B,0
1270  '              PUSH   BC         ; set up scroll
1280  '              PUSH   HL         ; preserve reg's
1290  '              PUSH   DE         ; on the stack
1300  '              CALL   &BC0B      ; get offset
1310  '              LD    DE,2
1320  '              ADD   HL,DE
1330  '              LD     A,(PAPER); get paper
1340  '              CALL   &BC05      ; update offset 6845
1350  '              LD     A,(PAPER)
1360  '              POP    DE
1370  '              POP    HL
1380  '              POP    BC
1390  '              CALL   &BC50      ; scroll col get
1400  '                                ; lined up
1410  '              EI                ; enable int
1420  '              RET
1430  ' OTHER        LD     H,0        ; set up reg's for
1440  '              LD     L,0        ; col scroll at end
1450  '              LD     D,0        ; of routine
1460  '              LD     E,24
1470  '              LD     B,1
1480  '              LD     A,(PAPER)
1490  '              PUSH   AF
1500  '              PUSH   BC
1510  '              PUSH   DE
1520  '              PUSH   HL
1530  '              CALL   &BC0B     ; get offset
1540  '              XOR    A
1545  '              LD     DE,2
1550  '              SBC    HL,DE
1560  '              LD     A,(PAPER)
1570  '              CALL   &BC05     ; send it to 6845
1580  '              POP    HL
1590  '              POP    DE
1600  '              POP    BC
1610  '              POP    AF
1620  '              CALL   &BC50     ; scroll left col
1630  '                               ; lined up
1640  '              EI
1650  '              RET
1660  '
1670  '
1680  '
1690  ' PAPER        BYTE   0
1700  ' END
1710  RETURN
```

```
F3 CD 99 BB CD 2C BC 32 67 9D DD 7E 00 FE 00 28 25 26 27
2E 00 16 27 1E 18 06 00 C5 E5 D5 CD 0B BC 11 02 00 19 3A
67 9D CD 05 BC 3A 67 9D D1 E1 C1 CD 50 BC FB C9 26 00 2E
00 16 00 1E 18 06 01 3A 67 9D F5 C5 D5 E5 CD 0B BC AF 11
02 00 ED 52 3A 67 9D CD 05 BC E1 D1 C1 F1 CD 50 BC FB C9
00
```

You can probably see that it would be possible to incorporate all these routines into one program, the routine checking the screen mode in use and setting up the registers for SW SCROLL and for the Screen Offset routine appropriately.

We've now seen how we can scroll the whole screen up and down, and the whole screen sideways. Using the Firmware routine called at address &BC5Ø, we can also scroll a given area of the screen, which I'll call the Scrolling Window, up or down. The next two routines that we'll examine scroll a small area of the screen sideways — a horizontal version of SW SCROLL. Material that is scrolled out of one side of the Scrolling Window is lost. However, to do these scrolls we must take a brief look at the way in which the Amstrad Video Memory is arranged, because we are going to have to work out a method of directly accessing the screen memory to accomplish the rest of the scrolls in this Chapter. All of the remaining scroll routines will still work on the character space as the basic unit of movement during a scroll. So, on to screen layout.

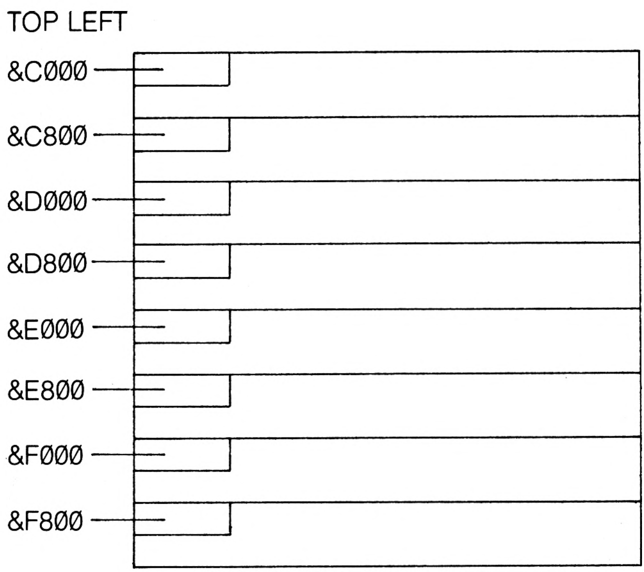
Each character on the screen, in any screen mode, is made up vertically of 8 Screen Lines. The screen as a whole is 25 rows deep, thus giving a total of 2ØØ screen lines in all modes. All screen modes use 16384 bytes of RAM, starting at address &CØØØ and finishing at address &FFFF. All the screen modes have displays that are 8Ø bytes wide. That is, one screen line, from left to right, takes up 8Ø bytes of screen RAM to define its contents. This explains the need for 16384 bytes of screen RAM. (2ØØ screen lines at 8Ø bytes per line.)

The exact layout of the screen memory with respect to character squares, etc. depends upon the screen mode in use.

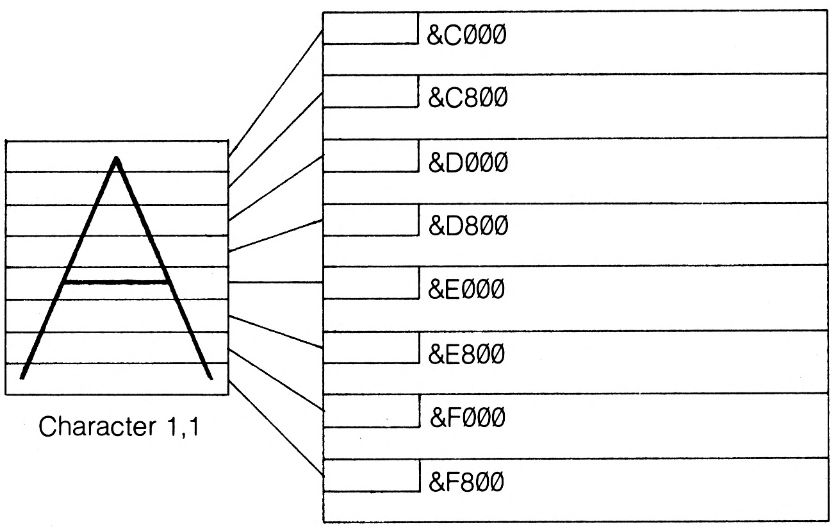
Mode 2

This is the simplest screen mode to use. Like the screen memory in all the other screen modes, the Video RAM is split into 8 2k blocks of memory, which after a mode change is arranged in the below fashion.

In Mode 2, each character is 1 byte wide; this is why we have an 8Ø column screen in this mode. Each bit of each byte corresponds to a screen pixel. Thus, the top of screen location 1,1 is, after a mode change, defined by the contents of &CØØØ. The character is made up of 8 screen lines, and these are taken from the other 7 2k blocks of memory. In this case, row 2 of the character is defined by the byte at &C8ØØ, row three by &DØØØ, row four by &D8ØØ and so on.



Screen RAM Layout

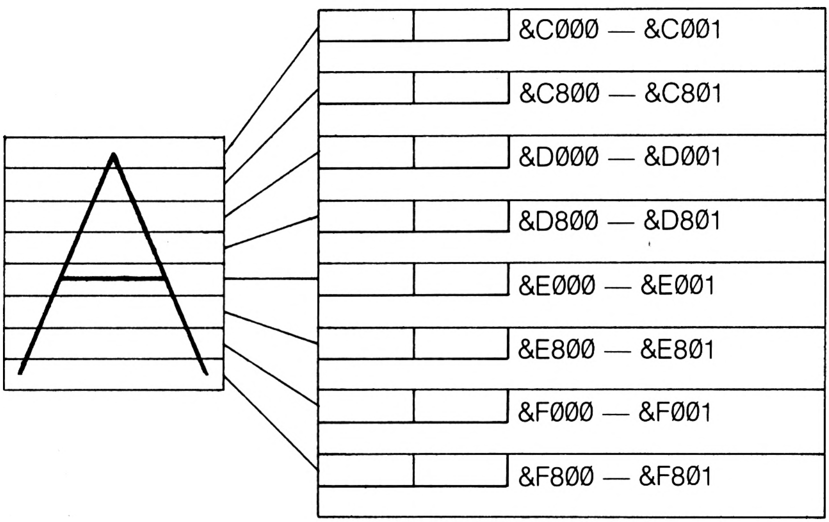


Mode 2 Character Mapping

If there are graphics on the screen, then the corresponding character positions will have certain bytes altered according to the graphics drawn. Once we know the address within the Screen RAM of the top row of the character in question, we can obtain the other 7 bytes that make up the definition of that character square by adding 2Ø48 to the first address repeatedly to get the addresses of the other screen rows. There is no colour information in Mode 2; if a pixel is to be displayed in the foreground colour, then the corresponding bit in the appropriate byte is set to 1. If it is to be set to the background colour, then the bit is set to Ø. The relationship between rows of the same character position — that is, them being separated from each other by 2Ø48 bytes, is the same in all screen modes.

Mode 1

Life gets a little more complicated here, due to the ability in this mode to display more than one foreground colour. Each character square is two bytes wide, thus explaining the fact that the Mode 1 screen is 4Ø columns wide. A Mode 1 character square is defined in the below fashion. Again, we're looking at the defining bytes for screen character location 1,1 after a Mode change.

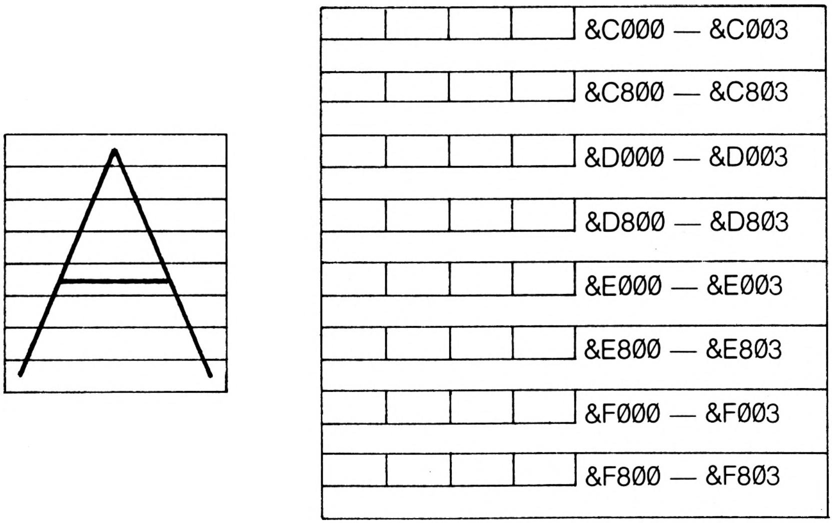


Each byte of screen RAM defines the colour status of the pixels, as well as the on or off state of them.

Each byte of screen RAM therefore defines the status of half the width of a character for a screen line. Each character square is thus defined by 16 bytes of Video RAM.

Mode 0

Because of the fact that there are 16 colours available in this screen mode, each character requires more bytes to define it and so each character is 4 bytes wide. This gives us the 2Ø column Mode 2 text screen. Immediately after a Mode Ø command, the mapping of Video RAM on to screen position 1,1 is as shown below.



Mode 0 — Screen RAM layout

Mode Ø therefore requires 32 bytes of Video RAM to define each character.

Well, we know that each screen line is 2Ø48 bytes apart from the next screen line of the same character. All we need now is some means of finding out the address in Video RAM of the top screen line of each character that we are interested in.

Fortunately, the nice chaps at Amsoft have solved this problem by giving us a Firmware routine to do the job. This is called at address &BC1A, and the character position is passed over to the routine in the HL register pair. The H register holds the X position and the L registers the Y position. Both X and Y are measured from Ø upwards, Ø,Ø being the top left corner of the screen.

The routine returns an address in the HL register pair. For Mode 2, this is the address of the top screen row of the character position concerned. For Modes Ø and 1 it is the address of the left most byte of the top screen row. Thus the address of the first byte of the second screen row will be at address (HL+2Ø48). At the heart of the scroll routines that we are about to see are the LDIR and LDDR block move instructions of the Z-8Ø CPU. For those of you not totally conversant with these instructions, I'll give a quick description of them and their use.

LDIR and LDDR

One method of transferring bytes around the computer memory might be to repeat a loop of instructions like the below a given number of times.

```
LD A,(HL)
LD (DE),A
INC HL
INC DE
```

This sequence is quite straight forward, and is able to do the transfer quite adequately. The problem is that with a lot of data to be transferred, this sequence, when often repeated, can take quite a lot of time. However, the Z-8Ø has built in block transfer instructions. LDIR is set up in the below fashion.

```
LD    HL,source address
LD    DE,Destination Address
LD    BC,no. of bytes
LDIR
```

The address in the HL register is the address from which bytes are to be transferred, and DE is the address to which the bytes are to be copied. The BC register contains the number of bytes to be transferred. The first byte transferred will go to the address in DE, the second to address DE+1, and so on. Both HL and DE are incremented between each individual transfer. Once the LDIR instruction starts executing, it will not finish until all the bytes are transferred. It is very much faster than doing the same job with individual Z-8Ø instructions, and so whenever you've got a lot of data to copy you should use this instruction or LDDR. LDDR does the same thing, except the DE and HL registers are decremented between each individual transfer.

The listings of these scroll routines are well annotated, and you should be able to follow what's happening in them by examining the listings and the above notes on the screen RAM arrangement for the different screen modes.

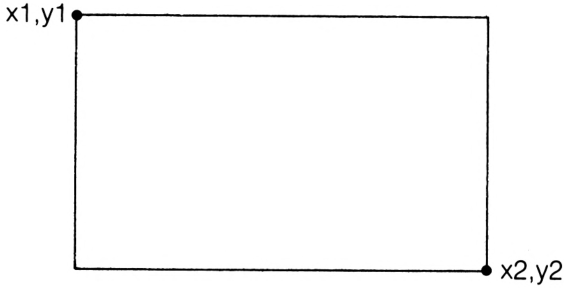
LSCROLL

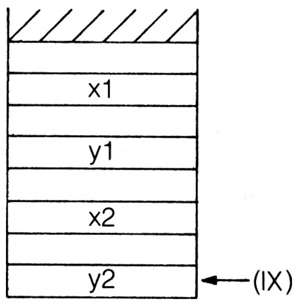
This routine will, in any mode, scroll a given area of the screen one character space to the left.

Any material that is scrolled out of the left edge of the Scrolling Window is lost. The routine is thus analogous to the SW SCROLL routine that is in the Firmware.

Entry Requirements: From BASIC: CALL address,x1,y1,x2,y2

Coordinates are from 1,1, this being the top left corner of the screen. From machine code, A=4 and IX points to a parameter block like the one shown.





Parameter Block for LSCROLL

Exit Conditions: All Corrupt.

Length: 169 Bytes.

LSCROLL

```
1000  MEMORY 39999
1010  MODE 1
1020  GOSUB 1070
1030  CLS : FOR I=1 TO 10 : PRINT "12345678901234567890" : NE
      XT
1040  CALL 40200,2,2,10,10
1050  FOR I=0 TO 200 : NEXT : GOTO 1040
1060  END
1070   ASSEMBLE
1080              ORG       40200
```

```
1090            CP      4
1100            RET     NZ           ; if <> 4 return
1110            CALL    &BC11        ; get screen mode
1120            LD      (MODE),A
1130            CALL    GWIDTH       ; no. of chars to move
1140            LD      (CHAR),A
1150            CALL    MODEC        ; adjust this value to
1160            LD      (CHAR),A     ; suit the mode in use
1170            XOR     A
1180            LD      (CHAR+1),A   ; set upper byte to 0
1190            LD      H,(IX+6)     ; get top left corner
1200            LD      L,(IX+4)     ; coordinates
1230            CALL    HEIGHT       ; no. of lines?
1240    OLOOP   PUSH    HL
1250            PUSH    BC
1260            CALL    MOVER        ; do the scroll
1270            POP     BC
1280            POP     HL
1285            CALL    PUTSP        ; fill right col
1290            INC     L            ; scroll next line
1300            DJNZ    OLOOP
1310            RET
1320    MOVER   DEC     L
1330            DEC     H            ; con coord for
1340            CALL    &BC1A        ; add calc
1350            ld      b,8          ; 8 bytes to move
1360            PUSH    HL
1370            POP     DE           ; add move to in DE
1380            INC     HL           ; get add to mve char
1390            CALL    FUDGE        ; byte from
1400    LOOP    PUSH    BC
1410            PUSH    DE
1420            PUSH    HL
1430            LD      BC,(CHAR)    ; No of hor bytes
1440            LDIR                 ; block move
1450            LD      BC,2048      ; next is 2048 on
1460            POP     HL
1470            ADD     HL,BC        ; next source address
1480            POP     DE
1490            PUSH    HL
1500            PUSH    DE
1510            POP     HL
1520            ADD     HL,BC
1530            PUSH    HL
1540            POP     DE          ; next dest add
1550            POP     HL
1560            POP     BC
1570            DJNZ    LOOP        ; do for 8 bytes
1580            RET
1590    MODEC   LD      A,(MODE)    ; adjusts hor no bytes
1600            CP      2           ; to move (mode)
1610            JR      NZ,MOD1
1620            LD      A,(CHAR)
1630            RET
1640    MOD1    CP      0
1650            JR      Z,MOD0
1660            LD      A,(CHAR)
1670            ADD     A           ; mode 1, 2 bytes width
1680            RET
1690    MOD0    LD      A,(CHAR)
1700            ADD     A
1710            ADD     A           ; mode 0, 4 bytes width
1720            RET
```

```
1730  ' FUDGE    LD      A,(MODE)   ; adjust source add
1740  '          CP      2          ; for char width
1750  '          RET     Z
1760  '          CP      1
1770  '          JR      NZ,MOV0
1780  '          INC     HL
1790  '          RET
1800  ' MOV0     INC     HL
1810  '          INC     HL
1820  '          INC     HL
1830  '          RET
1840  ' HEIGHT   PUSH    HL         ; calc lines
1850  '          LD      A,(ix)     ; to move
1860  '          LD      H,(IX+4)
1870  '          SUB     H
1880  '          INC     A
1890  '          LD      B,A
1900  '          POP     HL
1910  '          RET
1920  ' GWIDTH   LD      A,(IX+2)   ; no. of char. to
1925  '          PUSH    HL         ; moved hor
1930  '          LD      H,(IX+6)
1940  '          SUB     H
1950  '          INC     A
1955  '          POP     HL
1960  '          RET
2170  ' .PUTSP  PUSH     HL         ; fill in right
2175  '          LD      H,(IX+2)   ; col with space
2176  '          CALL    &BB75
2180  '         LD       A,32
2190  '         CALL     &BB5A
2200  '         POP      HL
2210  '         RET
2270  ' CHAR     WORD    00
2290  ' MODE     BYTE    0
2310  ' END
2320  RETURN
```

```
FE 04 C0 CD 11 BC 32 B0 9D CD 95 9D 32 AE 9D CD 5F 9D 32
AE 9D AF 32 AF 9D DD 66 06 DD 6E 04 CD 89 9D E5 C5 CD 38
9D C1 E1 CD A0 9D 2C 10 F3 C9 2D 25 CD 1A BC 06 08 E5 D1
23 CD 79 9D C5 D5 E5 ED 4B AE 9D ED B0 01 00 08 E1 09 D1
E5 D5 E1 09 E5 D1 E1 C1 10 E7 C9 3A B0 9D FE 02 20 04 3A
AE 9D C9 FE 00 28 05 3A AE 9D 87 C9 3A AE 9D 87 87 C9 3A
B0 9D FE 02 C8 FE 01 20 02 23 C9 23 23 23 C9 E5 DD 7E 00
DD 66 04 94 3C 47 E1 C9 DD 7E 02 E5 DD 66 06 94 3C E1 C9
E5 DD 66 02 CD 75 BB 3E 20 CD 5A BB E1 C9 12 00 01
```

Notes This routine is not protected against 'funny' parameters being passed over to it, so don't try passing ridiculously large x and y co-ordinates to it, or having y2 smaller than y1. You have been warned!!

The program cannot easily be relocated, and the bytes above are for address 40200. Note how we use a Firmware routine to recover the screen mode in use. The routine, called at &BC11, returns a value in the A register that corresponds to the screen mode. The routine then moves the RAM bytes according to the screen mode.

RSCROLL

This routine scrolls a defined area of the screen one character space to the right. Again, anything scrolled out of the right edge of the window is lost. The routine will work in any screen mode.

Entry Requirements: As for LSCROLL

Exit Conditions: As for LSCROLL

Length: 183 Bytes.

RSCROLL

```
1000  MEMORY 39999
1010  MODE 1
1020  GOSUB 1070
1030  CLS : FOR I=1 TO 10 : PRINT "12345678901234567890" : NE
      XT
1040  CALL 40600,2,2,10,10
1050  FOR I=0 TO 200 : NEXT : GOTO 1040
1060  END
1070   ASSEMBLE
1080  '          ORG       40600
1090  '          CP        4
1100  '          RET       NZ          ; if parm <> 4 ret
1110  '          CALL      &BC11       ; get screen mode
1120  '          LD        (MODE),A
1130  '          CALL      GWIDTH      ; no. of chars to move
1140  '          LD        (CHAR),A
1150  '          CALL      MODEC       ; adjust this value to
1160  '          LD        (CHAR),A    ; suit the mode in use
1170  '          XOR       A
1180  '          LD        (CHAR+1),A  ; set upper byte to 0
1190  '          LD        H,(IX+2)    ; get top right corner
1200  '          LD        L,(IX+4)    ; coordinates
1210  '          CALL      HEIGHT      ; calc no. lines
1220  ' OLOOP    PUSH      HL
1230  '          PUSH      BC
1240  '          CALL      MOVER       ; do the scroll
1250  '          POP       BC
1260  '          POP       HL
1270  '          CALL      PUTSP     ; fill right col spaces
1280  '          INC       L           ; scroll next line
1290  '          DJNZ      OLOOP
1300  '          RET
1310  ' MOVER    DEC       L
1320  '          DEC       H           ; con the coord for
1330  '          CALL      &BC1A       ; the address calc
1331  '          LD        A,(MODE)    ; now adjust accord to
1332  '          CP        2           ; the screen mode in use

1333  '          JR        Z,OK
1334  '          CP        1
1335  '          JR        Z,OK2
1340  '          INC       HL
1350  '          INC       HL
1360  ' OK2      INC       HL
1370  ' OK       LD        B,8         ; 8 bytes to move
1380  '          PUSH      HL
```

```
1390  '         POP    DE           ; add to move to in DE
1400  '         DEC    HL           ; get add to move char
1410  '         CALL   FUDGE        ; byte from
1420  ' LOOP    PUSH   BC
1430  '         PUSH   DE
1440  '         PUSH   HL
1450  '         LD     BC,(CHAR)    ; no. of hor bytes
1460  '         LDDR                ; block move
1470  '         LD     BC,2048      ;next char 2048 bytes on

1480  '         POP    HL
1490  '         ADD    HL,BC        ; next source address
1500  '         POP    DE
1510  '         PUSH   HL
1520  '         PUSH   DE
1530  '         POP    HL
1540  '         ADD    HL,BC
1550  '         PUSH   HL
1560  '         POP    DE           ; next destination add
1570  '         POP    HL
1580  '         POP    BC
1590  '         DJNZ   LOOP         ; do for 8 bytes
1600  '         RET
1610  ' MODEC   LD     A,(MODE)     ; adjusts hor no. bytes
1620  '         CP     2            ; to move for mode
1630  '         JR     NZ,MOD1
1640  '         LD     A,(CHAR)
1650  '         RET
1660  ' MOD1    CP     0
1670  '         JR     Z,MOD0
1680  '         LD     A,(CHAR)
1690  '         ADD    A            ; mode 1, 2 bytes
1700  '         RET
1710  ' MOD0    LD     A,(CHAR)
1720  '         ADD    A
1730  '         ADD    A            ; mode 0, 4 bytes
1740  '         RET
1750  ' FUDGE   LD     A,(MODE)     ; adjust source address
1760  '         CP     2            ; to suit char. width
1770  '         RET    Z
1780  '         CP     1
1790  '         JR     NZ,MOV0
1800  '         DEC    HL
1810  '         RET
1820  ' MOV0    DEC    HL
1830  '         DEC    HL
1840  '         DEC    HL
1850  '         RET
1860  ' HEIGHT  PUSH   HL           ; calc no. of lines
1870  '         LD     A,(ix)       ; that are to be moved
1880  '         LD     H,(IX+4)
1890  '         SUB    H
1900  '         INC    A
1910  '         LD     B,A
1920  '         POP    HL
1930  '         RET
1940  ' GWIDTH  LD     A,(IX+2)     ; gets no. of char.
1950  '         PUSH   HL           ; moved hor
1960  '         LD     H,(IX+6)
1970  '         SUB    H
1980  '         INC    A
1990  '         POP    HL
2000  '         RET
```

```
2010  ' .PUTSP   PUSH      HL          ; fill left
2020  '          LD       H,(IX+6)     ; col with space
2030  '          CALL     &BB75
2040  '          LD        A,32
2050  '          CALL      &BB5A
2060  '          POP       HL
2070  '          RET
2080  ' CHAR     WORD     00
2090  ' MODE     BYTE     0
2100  ' END
2110  RETURN
```

```
FE 04 C0 CD 11 BC 32 4E 9F CD 33 9F 32 4C 9F CD FD 9E 32
4C 9F AF 32 4D 9F DD 66 02 DD 6E 04 CD 27 9F E5 C5 CD C8
9E C1 E1 CD 3E 9F 2C 10 F3 C9 2D 25 CD 1A BC 3A 4E 9F FE
02 28 07 FE 01 28 02 23 23 23 06 08 E5 D1 2B CD 17 9F C5
D5 E5 ED 4B 4C 9F ED B8 01 00 08 E1 09 D1 E5 D5 E1 09 E5
D1 E1 C1 10 E7 C9 3A 4E 9F FE 02 20 04 3A 4C 9F C9 FE 00
28 05 3A 4C 9F 87 C9 3A 4C 9F 87 87 C9 3A 4E 9F FE 02 C8
FE 01 20 02 2B C9 2B 2B 2B C9 E5 DD 7E 00 DD 66 04 94 3C
47 E1 C9 DD 7E 02 E5 DD 66 06 94 3C E1 C9 E5 DD 66 06 CD
75 BB 3E 20 CD 5A BB E1 C9 12 00 01
```

Notes As with LSCROLL, no error trapping is carried out on the parameters passed over to the routine. The routine is not easily relocatable, and the bytes above are for address 40600.

The slowest part of both LSCROLL and RSCROLL is the PUTSP routine that fills in the column that has been scrolled 'in' to the window with spaces. This area of the program uses a Firmware routine. It should be possible to replace it with a section of code that directly writes '0's in to the approppropriate screen RAM addresses.

The below BASIC program will demonstrate the LSCROLL and RSCROLL routines, depending upon the addresses used.

```
100 MODE 1
110 REM add=40200 for LSCROLL
120 REM add=40600 for RSCROLL
130 add=40200
140 CLS:FOR I=1 TO 10
150 PRINT "12345678901234567890"
160 NEXT I
170 CALL add,2,2,10,10
180 FOR I=0 TO 200:NEXT I:REM delay
190 GOTO 170
```

The main problem with the routines given so far is that material scrolled out of one edge of the window is lost for ever. The final three scroll routines that we'll look at will get around this, providing a version of SW SCROLL that scrolls in material that has been scrolled out of the window, and versions of LSCROLL and RSCROLL that do the same type of job. These routines will work in any screen mode and will scroll both text and graphics, just like the previous routines.

LSCRF

This routine scrolls a defined area of the screen one character space to the left. Anything that is scrolled out of the left edge of the screen is scrolled in to the right edge. This can be very impressive for scrolling title pages to programs, scrolling headings, etc., especially if under AFTER or EVERY control from BASIC.

Entry Requirements: As for LSCROLL

Exit Conditions: All Registers Corrupt

Length: 385 bytes.

LSCRF

```
1000  MEMORY 39999
1010  MODE 1
1020  GOSUB 1070
1030  CLS : FOR I=1 TO 10 : PRINT "12345678901234567890" : NE
      XT : PEN 2 : LOCATE 2,4 : PRINT "▲▲▲JOE▲▲▲"
1040  CALL 41500,2,2,10,10
1050  FOR I=0 TO 200 : NEXT : GOTO 1040
1060  END
1070  |ASSEMBLE
1080  '           ORG     41500
1090  '           CP      4
1100  '           RET     NZ
1110  '           CALL    &BC11
1120  '           LD      (MODE),A
1130  '           CALL    GWIDTH
1140  '           LD      (CHAR),A
1150  '           CALL    MODEC       ; adjust this value to
1160  '           LD      (CHAR),A    ; suit the mode in use
1170  '           XOR     A
1180  '           LD      (CHAR+1),A  ; set upper byte to 0
1190  '           LD      H,(IX+6)    ; get top left corner
1200  '           LD      L,(IX+4)    ; coordinates
1210  '           CALL    HEIGHT
1220  ' OLOOP     PUSH    HL
1230  '           PUSH    BC
1240  '           CALL    CHGET       ; get char
1250  '           CALL    MOVER       ; do the scroll
1260  '           POP     BC
1270  '           POP     HL
1280  '           CALL    CHPUT       ; fill right with char
1290  '           INC     L           ; scroll next line
1300  '           DJNZ    OLOOP
1310  '           RET
1320  ' MOVER     DEC     L
1330  '           DEC     H           ; con the coord for
1340  '           CALL    &BC1A       ; add calc
1350  '           ld      b,8         ; 8 bytes to move
1360  '           PUSH    HL
```

```
1370 '          POP     DE
1380 '          INC     HL
1390 '          CALL    FUDGE
1400 ' LOOP     PUSH    BC
1410 '          PUSH    DE
1420 '          PUSH    HL
1430 '          LD      BC,(CHAR)
1440 '          LDIR
1450 '          LD      BC,2048
1460 '          POP     HL
1470 '          ADD     HL,BC       ; next source add
1480 '          POP     DE
1490 '          PUSH    HL
1500 '          PUSH    DE
1510 '          POP     HL
1520 '          ADD     HL,BC
1530 '          PUSH    HL
1540 '          POP     DE         ; next dest add
1550 '          POP     HL
1560 '          POP     BC
1570 '          DJNZ    LOOP       ; do for 8 bytes
1580 '          RET
1590 ' MODEC    LD      A,(MODE)
1600 '          CP      2
1610 '          JR      NZ,MOD1
1620 '          LD      A,(CHAR)
1630 '          RET
1640 ' MOD1     CP      0
1650 '          JR      Z,MOD0
1660 '          LD      A,(CHAR)
1670 '          ADD     A
1680 '          RET
1690 ' MOD0     LD      A,(CHAR)
1700 '          ADD     A
1710 '          ADD     A
1720 '          RET
1730 ' FUDGE    LD      A,(MODE)
1740 '          CP      2
1750 '          RET     Z
1760 '          CP      1
1770 '          JR      NZ,MOV0
1780 '          INC     HL
1790 '          RET
1800 ' MOV0     INC     HL
1810 '          INC     HL
1820 '          INC     HL
1830 '          RET
1840 ' HEIGHT   PUSH    HL         ; calc no. lines
1850 '          LD      A,(ix)     ; moved
1860 '          LD      H,(IX+4)
1870 '          SUB     H
1880 '          INC     A
1890 '          LD      B,A
1900 '          POP     HL
1910 '          RET
1920 ' GWIDTH   LD      A,(IX+2)   ; gets char to be
1930 '          PUSH    HL         ; moved hor
1940 '          LD      H,(IX+6)
1950 '          SUB     H
1960 '          INC     A
1970 '          POP     HL
1980 '          RET
1990 ' CHPUT    PUSH    HL
```

```
2000 '          PUSH    DE
2010 '          PUSH    BC
2020 '          PUSH    IX
2030 '          LD      H,(IX+2)    ; get col to fill
2040 '          DEC     H
2050 '          DEC     L           ; physical row/col
2060 '          LD      A,(MODE)    ; select routine
2070 '          CP      1           ; for the current screen

2080 '          JR      Z,PMOD1     ; mode
2090 '          CP      0
2100 '          JR      Z,PMOD0
2110 '          CALL    INIT        ; routine for mode 2
2120 ' CHPL     LD      A,(IX)      ; get from buffer
2130 '          LD      (HL),A      ; put in video RAM
2140 '          INC     IX          ; next buffer pos.
2150 '          ADD     HL,DE       ; byte of vid RAM
2160 '          DJNZ    CHPL        ; for 8 bytes
2170 ' POK      POP     IX          ; restore reg's
2180 '          POP     BC
2190 '          POP     DE
2200 '          POP     HL
2210 '          RET
2220 ' PMOD1    CALL    INIT        ; mode 1 rout
2230 ' P1L      CALL    HLPUT       ; each char 2 wide
2240 '          CALL    HLPUT       ; store 2 bytes in vid
2250 '          DEC     HL
2260 '          DEC     HL
2270 '          ADD     HL,DE       ; add next byte RAM
2280 '          DJNZ    P1L
2290 '          JR      POK
2300 ' PMOD0    CALL    INIT        ; mode 0, 4 bytes
2310 ' P0L      CALL    HLPUT       ; get 4 bytes
2320 '          CALL    HLPUT       ; row of character
2330 '          CALL    HLPUT
2340 '          CALL    HLPUT
2350 '          DEC     HL
2360 '          DEC     HL
2370 '          DEC     HL
2380 '          DEC     HL
2390 '          ADD     HL,DE       ; next char. row
2400 '          DJNZ    P0L
2410 '          JR      POK
2420 ' CHGET    PUSH    HL
2430 '          PUSH    BC
2440 '          PUSH    DE
2450 '          PUSH    IX
2460 '          DEC     H
2470 '          DEC     L
2480 '          LD      A,(MODE)    ; get mode, jump
2490 '          CP      1           ; to rout for
2500 '          JR      Z,GMOD1     ; the mode in use
2510 '          CP      0
2520 '          JR      Z,GMOD0
2530 '          CALL    INIT        ; mode 2, 1 byte
2540 ' CHGL     LD      A,(HL)
2550 '          LD      (IX),A
2560 '          INC     IX
2570 '          ADD     HL,DE
2580 '          DJNZ    CHGL
2590 ' CHOK     POP     IX          ; restore reg's
2600 '          POP     DE
2610 '          POP     BC
```

```
2620 '          POP      HL
2630 '          RET
2640 ' GMOD1   CALL     INIT          ; routine for mode 1
2650 ' M1L     CALL     HLGET         ; save 2 bytes of vid
2660 '         CALL     HLGET         ; for each char/line
2670 '         DEC      HL
2680 '         DEC      HL
2690 '         ADD      HL,DE         ; next VRAM add
2700 '         DJNZ     M1L
2710 '         JR       CHOK
2720 ' INIT    CALL     &BC1A         ; rout get add of
2730 '         LD       IX,TEMP       ; start char square
2740 '         LD       B,8           ; in HL and set reg
2750 '         LD       DE,2048
2760 '         RET
2770 '         DJNZ     M1L
2780 ' GMOD0   CALL     INIT          ; rout for mode 0
2790 ' M0L     CALL     HLGET         ; each char 4 bytes wid
e
2800 '         CALL     HLGET
2810 '         CALL     HLGET
2820 '         CALL     HLGET
2830 '         DEC      HL
2840 '         DEC      HL
2850 '         DEC      HL
2860 '         DEC      HL
2870 '         ADD      HL,DE
2880 '         DJNZ     M0L
2890 '         JR       CHOK
2900 ' HLGET    LD      A,(HL)      ; transfers byte from
2910 '          LD      (IX),A      ; video RAM to buff
2920 '          INC     HL
2930 '          INC     IX
2940 '          RET
2950 ' HLPUT    LD      A,(IX)      ; transfers a byte from
2960 '          LD      (HL),A      ; buffer to video RAM
2970 '          INC     HL
2980 '          INC     IX
2990 '          RET
3000 ' CHAR     WORD    00
3010 ' CHAR2    BYTE    0
3020 ' MODE     BYTE    0
3030 ' TEMP     RMEM    40
3040 ' END
3050 RETURN
```

```
FE 04 C0 CD 11 BC 32 74 A3 CD AC A2 32 71 A3 CD 76 A2 32
71 A3 AF 32 72 A3 DD 66 06 DD 6E 04 CD A0 A2 E5 C5 CD 06
A3 CD 4F A2 C1 E1 CD B7 A2 2C 10 F0 C9 2D 25 CD 1A BC 06
08 E5 D1 23 CD 90 A2 C5 D5 E5 ED 4B 71 A3 ED B0 01 00 08
E1 09 D1 E5 D5 E1 09 E5 D1 E1 C1 10 E7 C9 3A 74 A3 FE 02
20 04 3A 71 A3 C9 FE 00 28 05 3A 71 A3 87 C9 3A 71 A3 87
87 C9 3A 74 A3 FE 02 C8 FE 01 20 02 23 C9 23 23 23 C9 E5
DD 7E 00 DD 66 04 94 3C 47 E1 C9 DD 7E 02 E5 DD 66 06 94
3C E1 C9 E5 D5 C5 DD E5 DD 66 02 25 2D 3A 74 A3 FE 01 28
16 FE 00 28 22 CD 3A A3 DD 7E 00 77 DD 23 19 10 F7 DD E1
C1 D1 E1 C9 CD 3A A3 CD 69 A3 CD 69 A3 2B 2B 19 10 F5 18
EA CD 3A A3 CD 69 A3 CD 69 A3 CD 69 A3 CD 69 A3 2B 2B 2B
2B 19 10 ED 18 D2 E5 C5 D5 DD E5 25 2D 3A 74 A3 FE 01 28
16 FE 00 28 31 CD 3A A3 7E DD 77 00 DD 23 19 10 F7 DD E1
D1 C1 E1 C9 CD 3A A3 CD 61 A3 CD 61 A3 2B 2B 19 10 F5 18
```

```
EA CD 1A BC DD 21 75 A3 06 08 11 00 08 C9 10 E4 CD 3A A3
CD 61 A3 CD 61 A3 CD 61 A3 CD 61 A3 2B 2B 2B 2B 19 10 ED
18 C3 7E DD 77 00 23 DD 23 C9 DD 7E 00 77 23 DD 23 C9 12
00 00 01 30 C0 60 60 60 00 70 C0 60 60 60 60 30 C0 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00
```

Notes The routine is not easily relocated, and the bytes given above are for address 41500. As with all the scroll routines, the speed of scrolling is dependant upon the size of the window to be scrolled. However, the routine is still very fast, even with a large window.

RSCRF

This routine scrolls a defined area of the screen one character space to the right. Any material on the screen that is scrolled out of the right edge of the Scrolling Window is scrolled in at the left edge.

Entry Requirements: As for RSCROLL

Exit Conditions: All Registers Corrupt

Length: 399 Bytes.

```
RSCRF

1000  MEMORY 39999
1010  MODE 1
1020  GOSUB 1070
1030  CLS : FOR I=1 TO 10 : PRINT "12345678901234567890" : NE
      XT : LOCATE 2,4 : PEN 3 : PRINT"▲▲▲JOE▲▲▲"
1040  CALL 40200,2,2,10,10
1050  GOTO 1040
1060  END
1070   ASSEMBLE
1080  '            ORG      40200
1090  '            CP       4
1100  '            RET      NZ          ; if not 4 param, return

1110  '            CALL     &BC11       ; get screen mode
1120  '            LD       (MODE),A
1130  '            CALL     GWIDTH      ; no. of chars to move
1140  '            LD       (CHAR),A
1150  '            CALL     MODEC       ; adjust this value to
1160  '            LD       (CHAR),A    ; suit the mode in use
1170  '            XOR      A
1180  '            LD       (CHAR+1),A  ; set upper byte to 0
1190  '            LD       H,(IX+2)    ; get top right corner
1200  '            LD       L,(IX+4)    ; coordinates
1210  '            CALL     HEIGHT      ; work out no. of lines
1220  ' OLOOP      PUSH     HL
1230  '            PUSH     BC
1240  '            CALL     CHGET
1250  '            CALL     MOVER       ; do the scroll
1260  '            POP      BC
1270  '            POP      HL
1280  '            CALL     CHPUT      ; fill left col with char
```

```
1290 '          INC     L            ; scroll next line
1300 '          DJNZ    OLOOP
1310 '          RET
1320 ' MOVER    DEC     L
1330 '          DEC     H            ; con the coord for
1340 '          CALL    &BC1A        ; the add calc
1350 '          LD      A,(MODE)     ; adjust according to
1360 '          CP      2            ; the screen mode in use

1370 '          JR      Z,OK
1380 '          CP      1
1390 '          JR      Z,OK2
1400 '          INC     HL
1410 '          INC     HL
1420 ' OK2      INC     HL
1430 ' OK       LD      B,8          ; 8 bvtes to move
1440 '          PUSH    HL
1450 '          POP     DE           ; add to move to in DE
1460 '          DEC     HL           ; aet address move char
1470 '          CALL    FUDGE        ; bvte from
1480 ' LOOP     PUSH    BC
1490 '          PUSH    DE
1500 '          PUSH    HL
1510 '          LD      BC,(CHAR)    ; no. of hor bvtes
1520 '          LDDR                 ; block move
1530 '          LD      BC,2048      ; next part char 2048 on

1540 '          POP     HL
1550 '          ADD     HL,BC        ; next source address
1560 '          POP     DE
1570 '          PUSH    HL
1580 '          PUSH    DE
1590 '          POP     HL
1600 '          ADD     HL,BC
1610 '          PUSH    HL
1620 '          POP     DE          ; next dest add
1630 '          POP     HL
1640 '          POP     BC
1650 '          DJNZ    LOOP        ; do for 8 bvtes
1660 '          RET
1670 ' MODEC    LD      A,(MODE)    ; adjusts hor no. bvtes
1680 '          CP      2           ; move to suit mode
1690 '          JR      NZ,MOD1
1700 '          LD      A,(CHAR)
1710 '          RET
1720 ' MOD1     CP      0
1730 '          JR      Z,MOD0
1740 '          LD      A,(CHAR)
1750 '          ADD     A           ; mode 1, 2 bvtes
1760 '          RET
1770 ' MOD0     LD      A,(CHAR)
1780 '          ADD     A
1790 '          ADD     A           ; mode 0, 4 bvtes
1800 '          RET
1810 ' FUDGE    LD      A,(MODE)    ; adjust srce add
1820 '          CP      2           ; suit char in use
1830 '          RET     Z
1840 '          CP      1
1850 '          JR      NZ,MOV0
1860 '          DEC     HL
1870 '          RET
1880 ' MOV0     DEC     HL
1890 '          DEC     HL
```

```
1900  '          DEC     HL
1910  '          RET
1920  ' HEIGHT   PUSH    HL          : calc no. of lines
1930  '          LD      A,(ix)      : that are to be moved
1940  '          LD      H,(IX+4)
1950  '          SUB     H
1960  '          INC     A
1970  '          LD      B,A
1980  '          POP     HL
1990  '          RET
2000  ' GWIDTH   LD      A,(IX+2)    :no. of char to be
2010  '          PUSH    HL          : moved horizontallv
2020  '          LD      H,(IX+6)
2030  '          SUB     H
2040  '          INC     A
2050  '          POP     HL
2060  '          RET
5000  ' CHPUT    PUSH    HL
5010  '          PUSH    DE
5020  '          PUSH    BC
5030  '          PUSH    IX
5040  '          LD      H,(IX+6)     : qet col to fill
5050  '          DEC     H
5060  '          DEC     L            : phvsical row/col
5070  '          LD      A,(MODE)     : select corr rout
5080  '          CP      1            : for curr scrn
5090  '          JR      Z,PMOD1      : mode
5100  '          CP      0
5110  '          JR      Z,PMOD0
5120  '          CALL    INIT         : rout for mode2
5130  ' CHPL     LD      A,(IX)       : qet from buffer
5140  '          LD      (HL),A       : put in video RAM
5150  '          INC     IX           : next buffer pos.
5160  '          ADD     HL,DE        : next bvte of VRAM
5170  '          DJNZ    CHPL         : for 8 bvtes
5180  ' POK      POP     IX           : restore reqisters
5190  '          POP     BC
5200  '          POP     DE
5210  '          POP     HL
5220  '          RET
5230  ' PMOD1    CALL    INIT         : mode 1 routine
5240  ' P1L      CALL    HLPUT        ; char 2 bvtes wide
5250  '          CALL    HLPUT        : 2 bvtes in vRAM
5260  '          DEC     HL
5270  '          DEC     HL
5280  '          ADD     HL,DE        : add next bvte VRAM
5290  '          DJNZ    P1L
5300  '          JR      POK
5310  ' PMOD0    CALL    INIT         : mode 0, 4 bvtes
5320  ' P0L      CALL    HLPUT        ; qet 4 bvtes for each
5330  '          CALL    HLPUT        : row of character
5340  '          CALL    HLPUT
5350  '          CALL    HLPUT
5360  '          DEC     HL
5370  '          DEC     HL
5380  '          DEC     HL
5390  '          DEC     HL
5400  '          ADD     HL,DE        : start next char/row
5410  '          DJNZ    P0L
5420  '          JR      POK
5430  ' CHGET    PUSH    HL
5440  '          PUSH    BC
5450  '          PUSH    DE
```

```
5460  '         PUSH   IX
5470  '         DEC    H
5480  '         DEC    L
5490  '         LD     A,(MODE)      : get mode, and jump
5500  '         CP     1             : to correct rout for
5510  '         JR     Z,GMOD1       : the mode in use
5520  '         CP     0
5530  '         JR     Z,GMOD0
5540  '         CALL   INIT          : mode 2, 1 byte
5550  ' CHGL    LD     A,(HL)
5560  '         LD     (IX),A
5570  '         INC    IX
5580  '         ADD    HL,DE
5590  '         DJNZ   CHGL
5600  ' CHOK    POP    IX            : restore registers
5610  '         POP    DE
5620  '         POP    BC
5630  '         POP    HL
5640  '         RET
5650  ' GMOD1   CALL   INIT          : rout for mode1
5660  ' M1L     CALL   HLGET         : save 2 bytes VRAM
5670  '         CALL   HLGET         : for each char. line
5680  '         DEC    HL
5690  '         DEC    HL
5700  '         ADD    HL,DE         : next VRAM address
5710  '         DJNZ   M1L
5720  '         JR     CHOK
5730  ' INIT    CALL   &BC1A         : rout gets add of
5740  '         LD     IX,TEMP       : start of char sqr
5750  '         LD     B,8           : into HL sets up reg
5760  '         LD     DE,2048
5770            RET
5780  '         DJNZ   M1L
5790  ' GMOD0   CALL   INIT          : rout mode 0, with
5800  ' M0L     CALL   HLGET         : each char 4 bytes
5810  '         CALL   HLGET
5820  '         CALL   HLGET
5830  '         CALL   HLGET
5840  '         DEC    HL
5850  '         DEC    HL
5860  '         DEC    HL
5870  '         DEC    HL
5880  '         ADD    HL,DE
5890  '         DJNZ   M0L
5900  '         JR     CHOK
5910  ' HLGET   LD     A,(HL)        : trans a byte from
5920  '         LD     (IX),A        : VRAM to buffer
5930  '         INC    HL
5940  '         INC    IX
5950  '         RET
5960  ' HLPUT   LD     A,(IX)        : trans a byte from
5970  '         LD     (HL),A        : buffer to VRAM
5980  '         INC    HL
5990  '         INC    IX
6000  '         RET
6010  ' CHAR    WORD   00
6020  ' CHAR2   BYTE   0
6030  ' MODE    BYTE   0
6040  ' TEMP    RMEM   40
6050  ' END
6060 RETURN
```

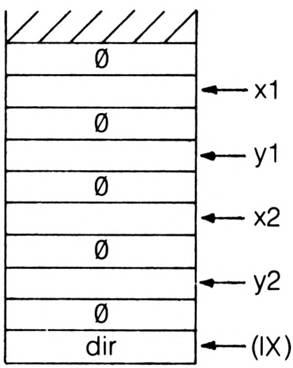
```
FE 04 C0 CD 11 BC 32 6E 9E CD A6 9D 32 6B 9E CD 70 9D 32
6B 9E AF 32 6C 9E DD 66 02 DD 6E 04 CD 9A 9D E5 C5 CD 00
9E CD 3B 9D C1 E1 CD B1 9D 2C 10 F0 C9 2D 25 CD 1A BC 3A
6E 9E FE 02 28 07 FE 01 28 02 23 23 23 06 08 E5 D1 2B CD
8A 9D C5 D5 E5 ED 4B 6B 9E ED B8 01 00 08 E1 09 D1 E5 D5
E1 09 E5 D1 E1 C1 10 E7 C9 3A 6E 9E FE 02 20 04 3A 6B 9E
C9 FE 00 28 05 3A 6B 9E 87 C9 3A 6B 9E 87 87 C9 3A 6E 9E
FE 02 C8 FE 01 20 02 2B C9 2B 2B 2B C9 E5 DD 7E 00 DD 66
04 94 3C 47 E1 C9 DD 7E 02 E5 DD 66 06 94 3C E1 C9 E5 D5
C5 DD E5 DD 66 06 25 2D 3A 6E 9E FE 01 28 16 FE 00 28 22
CD 34 9E DD 7E 00 77 DD 23 19 10 F7 DD E1 C1 D1 E1 C9 CD
34 9E CD 63 9E CD 63 9E 2B 2B 19 10 F5 18 EA CD 34 9E CD
63 9E CD 63 9E CD 63 9E CD 63 9E 2B 2B 2B 2B 19 10 ED 18
D2 E5 C5 D5 DD E5 25 2D 3A 6E 9E FE 01 28 16 FE 00 28 31
CD 34 9E 7E DD 77 00 DD 23 19 10 F7 DD E1 D1 C1 E1 C9 CD
34 9E CD 5B 9E CD 5B 9E 2B 2B 19 10 F5 18 EA CD 1A BC DD
21 6F 9E 06 08 11 00 08 C9 10 E4 CD 34 9E CD 5B 9E CD 5B
9E CD 5B 9E CD 5B 9E 2B 2B 2B 2B 19 10 ED 18 C3 7E DD 77
00 23 DD 23 C9 DD 7E 00 77 23 DD 23 C9 12 00 00 01 70 C0
C0 60 C0 E0 D0 60 E0 60 C0 60 70 C0 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Notes Again, the routine is not relocatable. The bytes above are for address 42ØØØ.

Both LSCRF and RSCRF can be demonstrated with the BASIC program that was listed for use with LSCROLL earlier in this Chapter.

CSCRF

This routine scrolls an area of the screen either up or down. Material scrolled out of either the top or bottom edge of the Scrolling Window is scrolled in at the other edge. The routine will work in any screen mode.



CSCRF Parameter Block

Entry Requirements: From BASIC, CALL address,x1,y1,x2,y2,dir where x1 =left edge of area
y1 =top of area
x2 =right edge of area
y2 =bottom row of area
dir =Ø – down scroll, dir =1 gives up scroll. Coordinates are from 1,1, this being the character square in the top left of the screen. If the routine is to be called from machine code, then IX must point to suitable parameter block and A must hold the value 5.

Exit Conditions: All Registers Corrupt.

Length: 914 Bytes, including buffer.

CSCRF

```
900   MODE 1
1000  MEMORY 39999
1010  GOSUB 1120
1020  CLS
1030  FOR I=1 TO 10 : PRINT "012345678901234567890123456789Ø"

1040  NEXT
1050  PEN 3 : LOCATE 2,3 : PRINT "**▲JOE▲**"
1060  G$=INKEY$ : IF G$="" THEN GOTO 1060
1070  IF ASC(G$)=240 THEN DIR=1
1080  IF ASC(G$)=241 THEN DIR=0
1090  CALL 41000,2,2,19,6,DIR
1100  GOTO 1060
1110  END
1120  ASSEMBLE
1130  '            ORG        41000
1140  '            CP         5
1150  '            RET        NZ          ; ret if not 5 param
1160  '            CALL       &BB99
1170  '            CALL       &BC2C
1180  '            LD         (PAPER),A   ; save text paper
1190  '            LD         A,(IX)
1200  '            CP         0           ; decide if up/down
1210  '            JR         Z,DOWN
1220  ' UP         CALL       GWID        ; widthg of window
1230  '            LD         H,(IX+8)
1240  '            LD         L,(IX+6)
1250  '            LD         A,1
1260  '            LD         (DIR),A
1270  '            CALL       GETC        ; save top line
1280  '            CALL       SCROLL      ; scroll up one
1290  '            CALL       GWID
1300  '            LD         H,(IX+8)
1310  '            LD         L,(IX+2)
1320  '            LD         A,0
1330  '            LD         (DIR),A
1340  '            CALL       GETC        ; print at bott
1350  '            RET
1360  ' DOWN       CALL       GWID
```

```
1370 '          LD      H,(IX+8)
1380 '          LD      L,(IX+2)
1385 '          LD      A,1           ; get bottom
1386 '          LD      (DIR),A       ; line....
1390 '          CALL    GETC          ; get bottom line
1400 '          CALL    SCROLL        ; scroll down 1
1410 '          CALL    GWID
1420 '          LD      H,(IX+8)
1430 '          LD      L,(IX+6)
1431 '          LD      A,0           ; prepare print
1432 '          LD      (DIR),A
1440 '          CALL    GETC          ; print at top
1450 '          RET
1460 ' GWID     PUSH    HL
1470 '          LD      A,(IX+4)
1480 '          LD      H,(IX+8)
1490 '          SUB     H
1500 '          LD      B,A
1510 '          INC     A
1520 '          LD      (WIDTH),A
1530 '          INC     B             ; width in to B reg
1540 '          POP     HL
1550 '          RET
1560 ' .GETC    PUSH    BC
1570 '          PUSH    DE
1580 '          PUSH    HL
1590 '          PUSH    IX            ; preserve the reg's
1600 '          LD      IX,BUFFER
1610 '          LD      B,8
1620 '          PUSH    BC
1630 '          DEC     H
1640 '          DEC     L
1650 '          CALL    &BC1A         ; get add of char
1660 '          LD      (NWID),BC     ; char width in NWID
1670 '          LD      DE,2048       ; set up DE
1680 '          POP     BC            ; rec no lines (8)
1690 ' OLOOP    PUSH    BC            ; save again
1700 '          LD      A,(WIDTH)     ; get no of char's
1710 '          LD      B,A
1720 '          PUSH    HL            ; save start add line

1730 ' OLOOP1   PUSH    BC            ; save no char's
1740 '          LD      BC,(NWID)     ; get bytes char wid
1750 ' LOOP     LD      A,(DIR)       ; get byte from VRAM
1760 '          CP      0             ; to buffer depending

1761 '          JR      Z,CHPUT       ; on value in DIR
1762 '          LD      A,(HL)
1763 '          LD      (IX),A
1770 ' CHOK     INC     IX            ; next byte...
1780 '          INC     HL            ; do a screen line of

1790 '          DJNZ    LOOP          ; char in wind
1800 '          POP     BC
1810 '          DJNZ    OLOOP1        ; for each char wind
1820 '          POP     HL
1830 '          ADD     HL,DE         ; get add lin VRAM
1840 '          POP     BC
1850 '          DJNZ    OLOOP         ; repeat for char
1860 '          POP     IX
1870 '          POP     HL
1880 '          POP     DE
1890 '          POP     BC            ; restore reg's
```

```
1900  '           RET
1902  ' CHPUT     LD       A,(IX)
1903  '           LD       (HL),A
1904  '           JR       CHOK
1910  ' SCROLL    LD       H,(IX+8)    ; scroll area
1920  '           LD       L,(IX+6)
1930  '           LD       D,(IX+4)
1940  '           LD       E,(IX+2)
1950  '           LD       B,(IX)
1960  '           LD       A,(PAPER)
1970  '           DEC      H
1980  '           DEC      L
1990  '           DEC      D
2000  '           DEC      E
2010  '           CALL     &BC50
2020  '           RET
2030  '
2040  ' NWID      WORD     00
2050  ' DIR       BYTE     0
2060  ' .WIDTH    BYTE     0
2070  ' BUFFER    RMEM     700
2080  ' PAPER     BYTE     0
2090  ' END
2100  RETURN


FE 05 C0 CD 99 BB CD 2C BC 32 B9 A3 DD 7E 00 FE 00 28 26
CD 87 A0 DD 66 08 DD 6E 06 3E 01 32 FB A0 CD 97 A0 CD DF
A0 CD 87 A0 DD 66 08 DD 6E 02 3E 00 32 FB A0 CD 97 A0 C9
CD 87 A0 DD 66 08 DD 6E 02 3E 01 32 FB A0 CD 97 A0 CD DF
A0 CD 87 A0 DD 66 08 DD 6E 06 3E 00 32 FB A0 CD 97 A0 C9
E5 DD 7E 04 DD 66 08 94 47 3C 32 FC A0 04 E1 C9 C5 D5 E5
DD E5 DD 21 FD A0 06 08 C5 25 2D CD 1A BC ED 43 F9 A0 11
00 08 C1 C5 3A FC A0 47 E5 C5 ED 4B F9 A0 3A FB A0 FE 00
28 17 7E DD 77 00 DD 23 23 10 F0 C1 10 E8 E1 19 C1 10 DD
DD E1 E1 D1 C1 C9 DD 7E 00 77 18 E7 DD 66 08 DD 6E 06 DD
56 04 DD 5E 02 DD 46 00 3A B9 A3 25 2D 15 1D CD 50 BC C9
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
ØØ ØØ
```

Notes The routine is non relocatable, due to the extensive use of subroutines. The bytes given above are for address 41ØØØ.

The last three routines all use a buffer area of RAM. This is used in the following way. For LSCRF and RSCRF, each screen row of the Scrolling Window is moved to the side in turn. Before it is scrolled, however, the area that will be lost by the scroll is copied in to the buffer area of memory. This will only be the amount of memory needed to define one character square, and so the most this will be will be 32 bytes for a Mode Ø character square. When the scroll is completed, we simply copy the buffer contents back in to the other edge of the screen row that's just been scrolled. As we are only copying screen RAM contents, the colour information and any graphics are retained, as well as character information. In CSCRF, we have to retain a whole screen row. This will be (80*8) bytes at maximum, assuming that some one may want to scroll the whole screen width. This is 80 bytes wide, and there are 8 bytes needed for each screen row in terms of screen lines. Each time a scroll is done, the screen row that would otherwise be lost is copied into this buffer area. Then, after the scroll, the contents of the buffer are copied in to the appropriate areas of screen RAM to restore the image at the other edge of the window.

5.
More Screen Routines

This Chapter is something of a mixed collection of routines for screen handling. So, we'll start by looking at methods of clearing the screen.

Clearing the Screen

The easiest way to do this from machine code is to call the routine at address &BC14. This will set the screen to ink Ø just like CLS. However, the text cursor will not be returned to the top left corner of the screen. Alternatively, the direct equivalent of CLS is

```
LD      A,12
CALL    &BB5A
```

which clears the text window and returns the cursor to the top left corner of the window. CLG, of course, also clears the screen, but the call to &BC14 does the job of CLG quite well.

However, all these methods of screen clearing are rather sudden, and its occasionally useful to have a routine that 'fades' the screen image gradually, rather than zapping it all at once. For example, you could use such a routine to fade out the title page of a program that you've written, giving your name the longest possible exposure! So, here are a couple of routines that offer this facility.

FCLS

This routine fades the screen into ink Ø. It is relocatable.

Entry Requirements: CALL address from BASIC or machine code.

Exit Conditions: AF,HL and DE are corrupt.

Length: 18 Bytes.

FCLS

```
1000  MEMORY 39999
1010  GOSUB 2000
1020  CALL 40200
1030  END
2000  |ASSEMBLE
2010  '              ORG    40200
2020  '              LD     E,254        ; first mask
2030  ' LOOP0        LD     HL,&C000     ; start of screen RAM
2040  ' LOOP1        LD     A,E          ; get mask into A
2050  '              AND    (HL)         ; mask with curr scrn
2060  '              LD     (HL),A       ; RAM byte, put back
2070  '              INC    HL           ; next byte of RAM
2080  '              LD     A,L          ; is address now 0000?

2090  '              OR     H
2100  '              JR     NZ,LOOP1     ; if not around again
2110  '              RL     E            ; rotate the mask
2120  '              JR     C,LOOP0      ; again if C is set
2130  '              RET
2140  ' EMD
2150  RETURN
```

```
1E FE 21 00 C0 7B A6 77 23 7D B4 20 F8 CB 13 38 F1 C9
```

Notes The routine works by repeatedly shifting a Ø through a byte that is otherwise set to hold all 1's. Each byte of the screen RAM is then ANDed with this mask, and put back in the screen RAM. This has the effect of gradually fading out the screen image. To speed things up a little, the end of screen memory is looked for by checking the HL register pair for zero; screen RAM ends at &FFFF and incrementing HL when it contains this value will give HL the contents 0000. Also, note the way in which the program checks to see if all 8 bits of the byte have been set to zero. We wait until the C flag is set to zero, thus indicating that the Ø from the byte has been rotated in to the C flag after being in each position in the byte.

SCRCLS

This routine clears the screen by simply scrolling the contents of the screen up or down by 25 lines. The routine is relocatable.

Entry Requirements: CALL address,dir from BASIC. dir=Ø indicates that a down scroll is wanted, dir=1 indicates an up scroll is required. From machine code, IX holds the address of the byte in memory holding the value of dir specifying the direction of scroll required. A=1.

Exit Conditions: All Corrupt.

Length: 48 Bytes, including temporary storage.

SCRCLS

```
1000  MEMORY 39999
1010  GOSUB 2000
1020  CALL 40200,1
1030  END
2000  |ASSEMBLE
2010  '              ORG      40200
2011  '              LD       A,(IX)
2012  '              LD       (DIR),A
2013  '              CALL     &BB99
2014  '              CALL     &BC2C      ; get text paper ink and
2015  '              LD       (PAPER),A; store it encoded
2020  '              LD       H,0        ; get the left edge
2030  '              LD       L,0        ; get the top row
2050  '              LD       E,24       ; bot lin to be scrolled
2051  '              CALL     &BC17      ; get the last column
2052  '              LD       D,B
2053  '              LD       B,25
2060  ' LOOP         PUSH     BC
2061  '              PUSH     DE
2062  '              PUSH     HL
2070  '              LD       A,(DIR)
2072  '              LD       B,A
2073  '              LD       A,(PAPER)
2080  '              CALL     &BC50
2082  '              POP      HL
2083  '              POP      DE
2090  '              POP      BC
2100  '              DJNZ     LOOP
2110  '              RET
2120  ' DIR          BYTE     0
2130  ' PAPER        BYTE     0
2140  ' EMD
2150  RETURN
```

```
DD 7E 00 32 36 9D CD 99 BB CD 2C BC 32 37 9D 26 00 2E 00
1E 18 CD 17 BC 50 06 19 C5 D5 E5 3A 36 9D 47 3A 37 9D CD
50 BC E1 D1 C1 10 EE C9 01 00
```

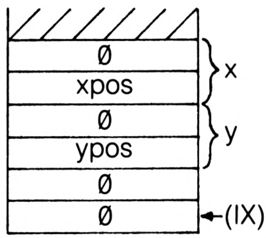
Notes The bytes above are for address 40200. When relocating the routine, don't forget to change the address of the temporary storage used. There is only one new ROM routine used in this program; the call to &BC17. We saw the scrolling routine in the last Chapter. This routine at &BC17 returns to us the last screen column and row available to us in the present screen mode. It is called SCR CHAR LIMITS by AMSOFT and on exit B holds the last screen column and C the last screen row available. We use this information to give the scrolling routine the correct number of columns to scroll for each screen mode.

When programming games routines, it's often useful to detect whether or not a character space on the screen is occupied by something else before you move another character in to it. The next two routines are designed to help out in this situation.

RDCHAR

This routine will return the ASCII code of any recognisable character at a specified screen position.

Entry Requirements: From BASIC, CALL address,x,y,@char% where x is the x coordinate, y is the y coordinate and char% is the variable which, on return, will hold the result of the call. From machine code, the IX register should point to a 6 byte parameter block, shown left, and A =3.



Parameter Block for RDCHAR

Exit Conditions: In BASIC, char% will hold the ASCII code if the character was recognisable, or 256 if the character was not recognisable. From machine code, location(IX) will hold the ASCII code for a 'legal' character and (IX+1) =Ø, or, for a character that hasn't been recognised by the system (IX+Ø) =Ø and (IX+1) =1. Alternatively, C =1 (Carry Flag set) and A =ASCII code for a legal character, otherwise A =Ø and the C flag is clear.

Length: 31 Bytes.

```
RDCHAR


1000   MEMORY 39999
1010   GOSUB 2000
1015   F%=0
1020   CALL 40200,10,10,@F%
1030   END
2000    ASSEMBLE
2010   '             ORG       40200
2015   '             CP        3
2016   '             RET       NZ          ; if wrong no. return
2020   '             LD        H,(IX+4)
2030   '             LD        L,(IX+2)
2040   '             CALL      &BB75       ; position cursor
```

```
2050  '           CALL    &BB60       ; get character
2055  '           LD      L,(IX+0)
2056  '           LD      H,(IX+1)
2060  '           JR      NC,NOCHAR   ; if char not known..
2061  '           LD      (HL),A      ; load ASCII code in to

2062  '           INC     HL          ; the variable
2063  '           LD      (HL),0
2064  '           RET
2065  ' NOCHAR    LD      (HL),0      ; if char not known,
2066  '           INC     HL          ; come here
2067  '           LD      (HL),1
2068  '           RET
2140  ' END
2150 RETURN
```

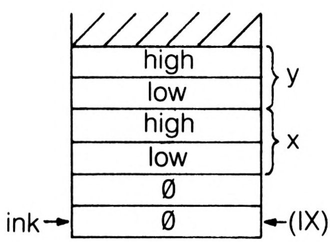
```
FE 03 C0 DD 66 04 DD 6E 02 CD 75 BB CD 60 BB DD 6E 00 DD
66 01 30 05 77 23 36 00 C9 36 00 23 36 01 C9
```

Notes The routine must be called with 3 parameters, otherwise an immediate return to BASIC is made. Remember that when using '@' to prefix variables, the variable used must have been previously declared. The most usual cause of 256 being returned is that a graphics PLOT or DRAW command has left a line or point in that particular character square, thus altering the image. A second cause of 256 being returned is that the pen and paper colour have been changed since that image was put on the screen. The way around this is to check the character position with each pen and paper combination. However, as we are usually just looking for the presence or absence of something, the routine is very useful.

A similar routine is called RDPOINT, but this gives the ink colour to be found at a particular pixel position on the screen.

RDPOINT

Returns to the user the ink colour of a specified pixel. The routine is relocatable.



RDPOINT Parameter Block

Entry Requirements: From BASIC, CALL address,x,y,@i% where x,y is the coordinate of the pixel and i% is the variable in which the ink is to be returned. If called from machine code, IX points to a parameter block like that shown. A =3.

Exit Conditions: All registers corrupt. If wrong number of parameters is passed, an immediate return to BASIC is made.

Length: 29 Bytes.

```
RDPOINT


1000   MEMORY 39999
1010   GOSUB 2000
1015   F%=0
1020   CALL 40200,10,10,@F%
1030   END
2000    ASSEMBLE
2010   '                ORG       40200
2015   '                CP        3
2016   '                RET       NZ ; if not 3 parameters, return
2020   '                LD        L,(IX+2)
2030   '                LD        H,(IX+3)
2040   '                LD        E,(IX+4)
2050   '                LD        D,(IX+5)
2060   '                CALL      &BBF0        ; get ink at DE,HL
2070   '                LD        L,(IX+0)
2080   '                LD        H,(IX+1)
2090   '                LD        (HL),A       ; store ink in var.
2100   '                INC       HL
2110   '                LD        (HL),0
2120   '                RET
2140   ' END
2150   RETURN

FE 03 C0 DD 6E 02 DD 66 03 DD 5E 04 DD 56 05 CD F0 BB DD
6E 00 DD 66 01 77 23 36 00 C9
```

Notes In this routine, and in RDCHAR, if you're only wanting the facility from machine code routines then it's easier to simply call the ROM routine directly, without putting the result returned in the appropriate variable. The two ROM routines used are as follows.

&BB60 This was discussed in Chapter 4.

&BBFO This routine is entered with DE holding the x coordinate of the pixel of interest and HL holding the y coordinate. On exit, the A register will hold the ink colour.

The next routine is another "decorative" one. It inverts the screen by swapping all the 1 bits in screen RAM to Ø and vice versa.

SCRINVERT

Changes the screen in any mode by complementing each byte of screen RAM. The routine is relocatable.

Entry Requirements: CALL address.

Exit Conditions: HL,AF are corrupt.

Length: 12 Bytes.

```
SCRINV


1000  MEMORY 39999
1010  GOSUB 2000
1020  CALL 40200
1030  END
2000  |ASSEMBLE
2010  '         org     40200
2020  '         LD      HL,&C000
2030  ' LOOP    LD      A,(HL)       ; get screen byte
2040  '         CPL                  ; complement byte
2050  '         LD      (HL),A       ; put it back in RAM
2060  '         INC     HL           ; next byte
2070  '         LD      A,L
2080  '         OR      H
2090  '         JR      NZ,LOOP      ; if HL not zero repeat
2100  '         RET
2110  ' END
2120  RETURN


21 00 C0 7E 2F 77 23 7D B4 20 F8 C9
```

Notes Each byte of screen RAM from &C000 to &FFFF is complemented. This simply means that each 0 is turned to 1 and each 1 is replaced with a 0. For example, the byte 10101010 when complemented is 01010101. The changes in colour that occur can be rather interesting, especially in Mode 0 with several colours on the screen at once. Calling it twice will restore the screen to it's original state.

The next routine is simple but heavy on memory use. It can be extremely useful when it is necessary to quickly alter the image that is displayed on the screen. We use the 16k of memory starting at address 26000 decimal as a temporary 'screen' which can hold a copy of the screen proper. We can draw an image on the screen, copy it in to this second screen, and then draw a second image. When we wish to display the contents of the second screen, we simply transfer the contents of RAM starting at 26000 to screen RAM.

SCRMOVE

This routine allows the use of an area of memory as a secondary 'screen' which can hold a copy of the current screen. The screen thus saved in RAM can be restored later in an almost instantaneous fashion.

Entry Requirements: CALL address,n from BASIC, where n specifies the 'direction' of data transfer. n=Ø transfers data from screen to RAM, and n=1 transfers data from RAM to screen. From machine code, IX points to a single byte of memory holding the value of n and A=1.

Exit Conditions: All registers corrupt.

Length: 21 Bytes, plus the area of memory from 26ØØØ decimal to 42384 inclusive.

SCRMVE

```
1000  MEMORY 24999
1010  GOSUB 1060
1020  CALL  25000,0
1030  CLS : INPUT "Press ENTER to restore screen",a$
1040  CALL  25000,1
1050  GOTO 1050 : REM prevents return of prompt
1060  |ASSEMBLE
1070  '          ORG      25000
1080  '          CP       1
1090  '          RET      NZ           ; return if wrong par
1100  '          LD       A,(IX)
1110  '          CP       0
1120  '          JR       Z,DOWN       ; if 0, scrn to RAM
1130  '          LD       DE,&C000     ; move RAM to screen
1140  '          LD       HL,26000
1150  '          LD       BC,&4000
1160  '          LDIR
1170  '          RET
1180  ' DOWN     LD       DE,26000
1190  '          LD       HL,&C000
1200  '          LD       BC,&4000
1210  '          LDIR
1220  '          RET
1230  ' END
1240  RETURN
```

```
FE 01 C0 DD 7E 00 FE 00 28 0C 11 00 C0 21 90 65 01 00 40
ED B0 C9 11 90 65 21 00 C0 01 00 40 ED B0 C9
```

Notes You will note that the area of memory used for temporary storage appears to be in a rather peculiar place. The reason for this is simple. The second screen area needs to be 16384 bytes long, and if we'd used memory much higher up in RAM then we would start overwriting the firmware jump blocks. The result of this is a rather catastrophic crash, as you might expect. Our machine code routine must therefore be in memory either between the end of our 'screen' and the firmware jump block or below address 26000. Within these constraints, though, the routine is relocatable.

Other points to note are as follows.

(i) Sensible results will only be obtained if the mode in use when the screen is restored is the same as that in use when the screen was saved.

(ii) An image should be saved before any scrolling occurs, and when you want to 'load' the screen from RAM it should be done immediately after a mode change. The scrolling operations alter the way in which screen RAM corresponds to particular screen positions.

SCRMOVE has the disadvantage of destroying the image on the screen when a transfer is made from our second 'screen' to the main screen. The next routine gets around this by swapping the contents of the two screens over.

SCRSWAP

This routine swaps the two 'screens' over, thus putting the current screen image in RAM starting at address 26000 and displaying the contents of RAM starting at 26000 on the screen. It is relocatable within the limits put forward in the notes for SCRMOVE.

Entry Requirements: CALL address.

Exit Conditions: All registers corrupt.

Length: 18 Bytes, plus the memory between address 26000 decimal and 42384 inclusive.

```
SCRSWP


1000   MEMORY 24999
1010   GOSUB 2000
1020   CALL  25000
1030   CLS : DRAW 100,100 : DRAW 100,200 : DRAW 0,300
1040   FOR I=0 TO 200 : NEXT
1050   CALL  25000
1060   GOTO 1040
2000   |ASSEMBLE
2010   '              ORG          25000
2020   '              LD           DE,26000     ; initialise registers
2030   '              LD           HL,&C000
2040   ' LOOP         LD           C,(HL)       ; get byte from screen
2050   '              LD           A,(DE)       ; get from temp store
2070   '              LD           (HL),A       ; next inst swap bytes
2071   '              LD           A,C
```

```
2072  '           LD       (DE),A
2080  '           INC      HL          ; next bytes pointed to
2090  '           INC      DE
2100  '           LD       A,L         ; HL is zero when all
2110  '           OR       H           ; screen done so check
2120  '           JR       NZ,LOOP
2130  '           RET
2140  ' END
2150  RETURN
```

```
11 90 65 21 00 C0 4E 1A 77 79 12 23 13 7D B4 20 F5 C9
```

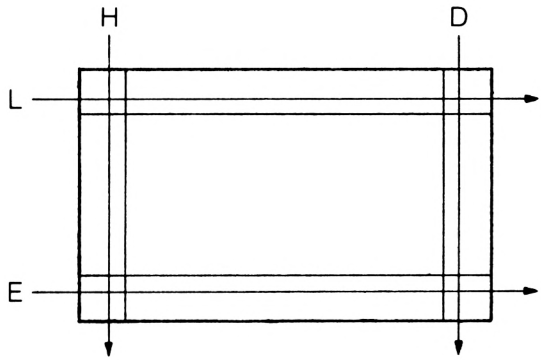
Notes The swapping of the two screens leads to a 'cross fade' type effect that can be obtained from slide projectors. The fade could be slowed down by inserting a delay loop in the machine code listing above. The below BASIC program demonstrates the machine code routine, which I've assumed is at address 25000 decimal.

```
100  FOR I=1 TO 20: PRINT "Hello There":NEXT I
110  CALL 25000
120  CLS: DRAW 100,100:DRAW 100,200:DRAW 0,300
130  FOR I=0 TO 300:NEXT:REM time delay
140  CALL25000
150  GOTO 130
```

Fill Routines

You may remember how in Chapter 3 I listed a routine for drawing rectangles to the screen, with an option of them being filled or open. There are a variety of ways in which an area of screen can be filled with a colour, and later in this section we'll see a general purpose routine for filling a horizontal line, which can be the basis of general purpose fill routines. Before we look at this, however, a brief examination of the resident fill routines available to us might be of interest.

SCR FILL BOX, called at &BC44, will fill an area of screen with a specified colour. However, there are limitations in the resolution of the filled area, as its limits are specified in terms of character squares. On entry, A holds the encoded ink colour, which can be obtained from the normal ink number by the routine at address &BC2C. This was detailed in Chapter 4. HL and DE specify the area to be filled in the below fashion.



SCR FLOOD BOX gives us more resolution, but is a little more difficult to use. It is called at address &BC47 and fills the specified area with the ink colour whose encoded value is in the C register. The limits of the area to be filled are given in terms of screen RAM addresses. This usually involves us in the task of converting pixel or character locations in to screen addresses. There are ROM routines to allow us to do this, as we saw in Chapter 4. However, I'm not going to go into more detail here. Suffice to say that the HL pair holds the address of the top left corner of the area to be filled in, D holds the width of the area to be filled in, in bytes, and E holds the height of the area in screen lines.

In all modes, the screen is 200 screen lines high, and 80 bytes wide, hence explaining the presence of 16384 bytes of screen RAM (80*200). In Mode 0, each character is 4 bytes wide, in Mode 1 it is two bytes wide and in Mode 2 each character is only 1 byte wide. Thus to fill the whole screen in in a particular colour, which we're assuming is in C, we'd execute code like the below.

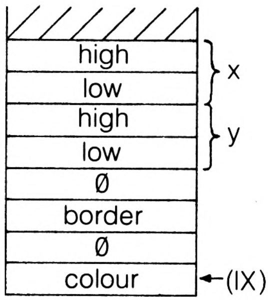
```
LD      HL,&C000 ; start of screen RAM-top left of screen
LD      D,80
LD      E,200
CALL    &BC47
RET
```

However, both these routines are rather unintelligent, in that we have to specify the borders of the area to be filled in, and this can be rather tedious, especially if we're trying to fill a non rectangular shape. What is required is a routine that fills a screen line to a particular specified border colour with a specified ink colour. Each screen line within a shape can then be filled in. After we've examined the machine code, we'll see how it can be used to fill shapes such as triangles or circles.

LINEFILL

This routine fills a single screen line between two 'border' pixels, the colour of which can be specified by the user. The colour in which the line is filled is also specified by the user. If you wish to relocate the routine, then it will be necessary to alter the address of 'RIGHT'. The bytes specified below are for address 40200 decimal.

Entry Requirements: From BASIC, CALL address,x,y,border,colour where x,y is the position to start the fill at, border is the ink colour to mark the limit of the fill and colour is the ink in which the line is to be filled. From machine code, IX points to a parameter block like that shown, and A=4.



Exit Conditions: If the number of parameters is incorrect then an immediate return to BASIC is made. All the registers are corrupt.

Length: 91 Bytes.

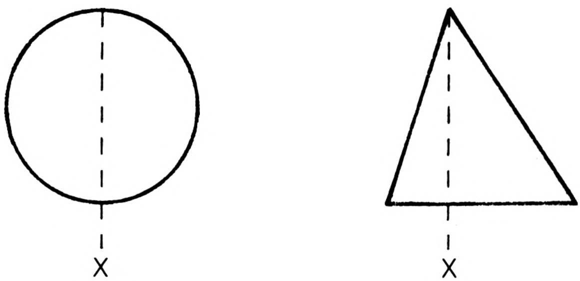
```
LNFILL


1000   MEMORY 39999
1010   GOSUB 2000
1020   MOVE 100,0 : DRAW 100,600 : MOVE 200,0 : DRAW 200,600
1030   CALL 40200,150,200,1,1
1040   END
2000   ASSEMBLE
2010   '            ORG          40200
2020   '            CP           4
2030   '            RET          NZ
2070   '            LD           L,(IX+4)
2080   '            LD           H,(IX+5)
2090   '            LD           E,(IX+6)
2095   '            LD           D,(IX+7)
2100   '            PUSH         HL
2110   '            PUSH         DE
2120   ' LOOP       PUSH         HL          ; start scan to right
```

```
2130  '         PUSH     DE
2140  '         CALL     &BBFØ      ; get ink of pixel
2150  '         POP      DE
2160  '         POP      HL
2170  '         CP       (IX+2)     ; is it border colour?
2180  '         JR       Z,OUTR     ; if so, out
2185  '         LD       A,3        ; if D=3 then we're out

2186  '         CP       D          ; of screen, so get out

2187  '         JR       Z,OUTR
2190  '         INC      DE         ; next pixel
2200  '         JR       LOOP
2210  ' .OUTR   DEC      DE         ; back one pixel
2220  '         LD       (RIGHT),DE; store it away
2230  '         POP      DE
2240  '         POP      HL
2250  ' LOOP2   PUSH     HL         ; now scan to the left
2260  '         PUSH     DE
2270  '         CALL     &BBFØ      ; get pixel colour
2280  '         POP      DE
2290  '         POP      HL
2300  '         CP       (IX+2)     ; is it border?
2310  '         JR       Z,OUTL
2320  '         DEC      DE         ; next pixel
2321  '         LD       A,E        ; if DE=Ø then we're
2322  '         OR       D          ; out of screen
2323  '         JR       Z,OUTL     ; for this line so out
2330  '         JR       LOOP2
2340  ' .OUTL   INC      DE         ; next pixel
2350  '         LD       L,(IX+4)   ; Y coordinate into HL
2360  '         LD       H,(IX+5)
2365  '         PUSH     HL         ; save it
2370  '         CALL     &BBCØ
2375  '         LD       A,(IX+Ø)   ; get the ink to fill
2376  '         CALL     &BBDE      ; line and set graph
2377  '                             ; colour
2380  '         POP      HL
2390  '         LD       DE,(RIGHT); get the right X coord

2410  '         CALL     &BBF6      ; draw the line
2420  '         RET                 ; done.
2430  ' RIGHT   WORD     ØØ
2440  ' END
2450  RETURN
```

```
FE Ø4 CØ DD 6E Ø4 DD 66 Ø5 DD 5E Ø6 DD 56 Ø7 E5 D5 E5 D5
CD FØ BB D1 E1 DD BE Ø2 28 Ø8 3E Ø3 BA 28 Ø3 13 18 EC 1B
ED 53 61 9D D1 E1 E5 D5 CD FØ BB D1 E1 DD BE Ø2 28 Ø7 1B
7B B2 28 Ø2 18 ED 13 DD 6E Ø4 DD 66 Ø5 E5 CD CØ BB DD 7E
ØØ CD DE BB E1 ED 5B 61 9D CD F6 BB C9 98 ØØ
```

Notes The method of operation of the routine is very simple, and use is made of the Amstrad firmware routines to allow use in all screen modes. It first searches to the right until either the x coordinate is greater than 75Ø odd or until a pixel in the appropriate border colour is found. The x position of this is then stored in the two byte variable 'RIGHT'. The routine then resets the x position to the start position. The search is then made in the left direction, until either a border pixel is found or until the x coordinate is equal to zero. The line is then filled in the required colour by a call to the line drawing routine. The colour that was selected for filling in the line will be the graphics pen colour after the fill has finished.

So, we can fill a line. What about real shapes? Well, it's quite easy. We choose an x coordinate that corresponds to the biggest y coordinate associated with a particular shape. This means that fill operations on complex shapes may have to be carried out in several steps. To make this clearer, look at the two diagrams below.



Circle **Triangle**

The fill is then carried out with a FOR...NEXT loop that alters the y coordinate from the lowest y coordinate on that x line to the highest y coordinate. The below BASIC program demonstrates this in action. I have assumed that the machine code routine is at address 4Ø2ØØ.

```
1ØØ  MODE 1: REM set screen mode
11Ø  PLOT Ø,Ø,1:REM move to Ø,Ø, set graphics
12Ø  REM pen to 1
13Ø  DRAW 1ØØ,1ØØ:DRAW 2ØØ,Ø:DRAW Ø,Ø:REM draw triangle
14Ø  FOR y=1 TO 99:REM y coordinate loop
15Ø  CALL 4Ø2ØØ,1ØØ,y,1,2
16Ø  NEXT
```

There are disadvantages with this routine, but it is still rather useful. The main problem is speed, and so is best used with smaller shapes or areas that need filling. This routine can be used with others to fill in the bulk of a shape, leaving the fiddly areas of the shape to the routine above.

GPEN

This simple routine just specifies the graphics pen and paper colour to be used. It is relocatable. The graphics paper chosen only comes into action after a CLG is executed.

Entry Requirements: BASIC CALL address,pen,paper where pen is the ink colour to be used for the graphics pen and paper is the colour to be used for the graphics paper. For machine code, IX points to a parameter block like that shown and A =2.

GPEN Parameter Block

Exit Conditions: All registers corrupt.

Length: 16 Bytes.

GPEN

```
1000  MEMORY 39999
1010  GOSUB 2000
1020  MODE 1 : PLOT 0,0,1
1030  DRAW 100,100 : CALL 40200,2,3 : DRAW 200,200
1040  END
2000  |ASSEMBLE
2010  '         ORG     40200
2020  '         CP      2    ; if not 2 parameters
2030  '         RET     NZ   ; return
2040  '         LD      A,(IX+0)
2050  '         CALL    &BBE4      ; change graphics paper
2060  '         LD      A,(IX+2)
2070  '         CALL    &BBDE      ; change graphics ink
2080  '         RET
2090  ' END
2100  RETURN
```

```
FE 02 C0 DD 7E 00 CD E4 BB DD 7E 02 CD DE BB C9
```

Notes The graphics pen colour comes into play immediately, the paper at the next CLG.

We'll now look at how we can move images around the screen, using machining code routines. The rest of this Chapter will be concerned with moving characters across the screen in any screen mode that you like. We will be using the routines resident in the Amstrad Firmware to print the characters to the screen, but this can still give good results. After looking briefly at the techniques that are involved, I'll present a program that moves a character around the screen. This provides the Amstrad with a simple "pseudo Sprite" for general purpose moving graphics. I'll then finish with a look at multicoloured characters, and how these can be moved around.

Moving Characters

The main operations that need to be carried out when we're moving characters around the screen are as follows:

(i) The character at the old position must be erased from the screen.

(ii) The x and y coordinates of the character must be updated to the new position.

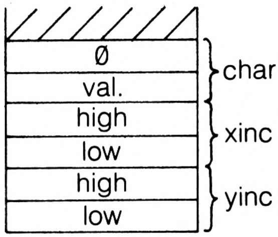
(iii) The character must be printed at the new position.

The smoothness of the resulting movement depends upon two factors. These are the amounts by which the x and y coordinates are altered and the frequency at which the position of the character is altered. Smoother movement will be obtained if the character is moved by only 1 or 2 pixels at a time than if we move it whole character squares at a time. Similarly, a rapid updating of the position of the character will give smoother motion. With regard to erasing the character, the most obvious way in which this can be done is to simply overprint the character with a space. However, this can cause problems if the character is moving across a background that has another image on it, as the other image will be erased as well. So, we'll not be using that technique. A second method is to use the XOR graphics mode. Without going into details, this will cause an image to disappear from the screen if it is printed twice to exactly the same place. There are some disadvantages with it, however; although this method leaves the background image on the screen, while the character of interest is being moved over it there can be some changes in colour of the background image. However, it is probably the simplest way of doing things that works reasonably well. Of course, you could always redraw the background after every move, but this would tend to slow things down rather a lot. So, let's look at a program to move around single coloured characters using XOR.

MOVECHAR

This routine moves a specified user defined character, or normal character, from one screen position to another instantly. At the heart of the program is a table of information, called a Shape Table, that holds information on each character that you will want to move during the program. The program can be relocated, provided that the address of 'TEMP' and the Shape Table are suitably altered as well.

Entry Requirements: From BASIC, CALL address,char,xinc,yinc where 'char' is the entry number in the Shape Table of the character that is to be moved, xinc is the alteration to be made to the x coordinate of the character position and yinc is the change to be made to the y coordinate of the character position. These alterations can be either positive or negative. The value of char must be greater than Ø. If called from machine code then A=3 and IX points to a parameter block like that shown. A Shape Table must also be present, as we'll soon see.



MOVECHAR Parameter Block

Exit Conditions: All Corrupt. The graphics pen colour will be that of the character drawn, the graphics mode will be 'Force' or 'Absolute' mode and the graphics cursor will be at the final position of the character.

Length: 127 Bytes, not including Shape Table.

MVECHR

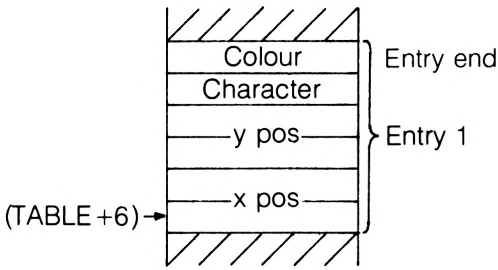
```
900   MODE 1
1000  MEMORY 39999
1010  GOSUB 2000
1016  J=1
1017  MOVE 100,100 : PRINT CHR$(23)+CHR$(1) : TAG : PRINT CHR
      $(234); : TAGOFF
1025  FOR I=1 TO 300 : CALL 42000,1,J,0 : NEXT
1030  END
2000   ASSEMBLE
2010  '         ORG     42000
2020  '         CP      3
2030  '         RET     NZ
2050  '         LD      B,(IX+4)
2060  '         PUSH    IX
2070  '         LD      IX,TABLE     ; start of shapetbl
2080  ' LOOP    INC     IX           ; get entry
2090  '         INC     IX
2100  '         INC     IX
2110  '         INC     IX
2120  '         INC     IX
2130  '         INC     IX
2140  '         DJNZ    LOOP
2145  '         LD      (TEMP),IX    ; store start entry
```

```
2150  '          LD      E,(IX)
2160  '          LD      D,(IX+1)       ; get x coordinate
2170  '          LD      L,(IX+2)       ; get y coordinate
2180  '          LD      H,(IX+3)
2190  '          PUSH    HL
2200  '          PUSH    DE
2210  '          CALL    &BBC0          ; move to the position

2220  '          LD      A,(IX+5)       ; set the colour
2230  '          CALL    &BBDE
2240  '          LD      A,1            ; set the XOR graphics

2250  '          CALL    &BC59          ; mode
2260  '          LD      A,(IX+4)
2270  '          CALL    &BBFC          ; print the character
2280  '          POP     DE
2290  '          POP     HL
2300  '          POP     IX             ; recover IX
2310  '          PUSH    HL
2320  '          PUSH    DE
2330  '          POP     HL
2340  '          LD      C,(IX+2)       ; add the increment to

2350  '          LD      B,(IX+3)       ; to x coordinate
2360  '          ADD     HL,BC
2370  '          PUSH    HL
2380  '          POP     DE             ; get it into DE
2381  '          PUSH    IX
2382  '          LD      IX,(TEMP)
2383  '          LD      (IX),E         ; store in table
2384  '          LD      (IX+1),D
2385  '          POP     IX
2390  '          LD      C,(IX+0)       ; add the increment to

2400  '          LD      B,(IX+1)       ; the y coordinate
2410  '          POP     HL
2420  '          ADD     HL,BC
2421  '          LD      IX,(TEMP)
2422  '          LD      (IX+2),L       ; and store in table
2423  '          LD      (IX+3),H
2430  '          CALL    &BBC0          ; move to new loc
2450  '          LD      A,(IX+4)       ; print the character
2460  '          CALL    &BBFC
2470  '          LD      A,0            ; set 'force' graph
2480  '          CALL    &BC59          ; mode.....
2490  '          RET                    ; and finish.
2495  ' TEMP     WORD    00
2500  ' TABLE    WORD    00             ; shape table
2510  '          WORD    00             ; is a dummy one
2520  '          WORD    00
2530  '          WORD    100
2540  '          WORD    100
2550  '          BYTE    234
2560  '          BYTE    1
2570  ' END
2580 RETURN
```

```
FE 03 C0 DD 46 04 DD E5 DD 21 91 A4 DD 23 DD 23 DD 23 DD
23 DD 23 DD 23 10 F2 DD 22 8F A4 DD 5E 00 DD 56 01 DD 6E
02 DD 66 03 E5 D5 CD C0 BB DD 7E 05 CD DE BB 3E 01 CD 59
BC DD 7E 04 CD FC BB D1 E1 DD E1 E5 D5 E1 DD 4E 02 DD 46
```

```
03 09 E5 D1 DD E5 DD 2A 8F A4 DD 73 00 DD 72 01 DD E1 DD
4E 00 DD 46 01 E1 09 DD 2A 8F A4 DD 75 02 DD 74 03 CD C0
BB DD 7E 04 CD FC BB 3E 00 CD 59 BC C9 00 00 00 00 00 00
00 00 64 00 64 00 EA 01
```

Notes The program relies on the presence of a Table of data which holds information on each character that the programmer is going to move using this routine. This Table, called the Shape Table, has a 6 byte entry for each character, and the CALL to this routine specifies, with the 'char' parameter, the Entry in the Shape Table containing details of the character to be moved rather than the ASCII code of the character itself. A typical entry in the Table is included in the above listing, and a more detailed examination of a typical entry is given below.



Shape Table Entry for MOVECHAR

An entry for each character is prepared before any machine code routine is called by POKEing appropriate values into the relevant Table entry, remembering that the first 6 bytes of the Table, that correspond to Entry Ø, are blank. The Table holds the initial x and y position of the character, the ASCII code of the character to be printed and the colour in which it is to be printed. The x and y entries will be updated whenever the position of the character is changed. By storing the information about the characters to be moved in this way the routine is made as generally useful as possible.

The program itself uses XOR to erase the character from one place and redraw it in another. The latter position is worked out by adding the x increment (xinc) to the current x coordinate and by adding the y increment (yinc) to the current y coordinate. This updated position is then stored in the Shape Table. This works perfectly well on all occasions except the very first appearance of the character on the screen; a 'shadow' of the character is left on the screen. A moment or two's thought will reveal the answer to this problem.

When the routine is entered, it first erases the last position of the character using XOR. However, if there is nothing there to erase, an image will be left on the screen. Subsequent operations will work perfectly. This is easily remedied by using a line of BASIC like the one below to initially position each character at it's start position.

```
PLOT 1000,1000,1:TAG:PRINT CHR$(23)+
CHR$(1);:MOVE startx,starty:PRINT CHR$(character);:TAGOFF
```

The PLOT 1000,1000 statement sets the graphics PEN colour to 1, and CHR$(23) followed by CHR$(1) sets up the graphics XOR mode. If you had 100,100 in the Shape Table entry as the start point for a given character, and 1 as the colour, and 254 as the ASCII code of the character of interest in the Table, the above line of BASIC would be:

```
PLOT 1000,1000,1:PRINT CHR$(23)+CHR$(1);:TAG:
MOVE 100,100:PRINT CHR$(254);:TAGOFF
```

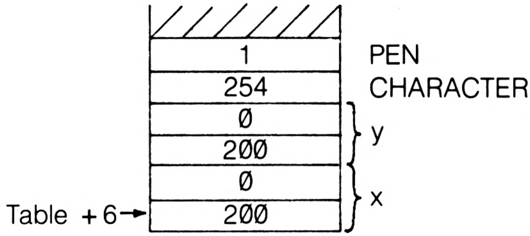
The below demonstration program assumes that the above bytes, including the Shape Table, is at address 42000.

```
100  MODE 1:PRINT CHR$(23)+CHR$(1)
110  PLOT 1000,1000,1:TAG
120  MOVE 100,100:PRINT CHR$(234);:TAGOFF
130  yinc=1:xinc=1
140  FOR I=1 TO 50:NEXT: REM delay loop
150  CALL 42000,1,xinc,yinc
160  GOTO 140
```

If you wish to re-run this program, then you'll have to restore the start x and y coordinates in the Shape Table to their initial values by POKEs, as they will have been modified during the running of the program.

When you are using this routine, a couple of things will become apparent. The first is that should you try and move the character too quickly, it will appear to 'roll'. This is due to interaction between the frequency of movement of the character and the rate at which the screen image is refreshed by the computer. The answer is simple; slow down, using either a delay loop in BASIC or machine code or a CALL &BD19, which will wait until the next screen frame has been drawn before carrying on with the program. Secondly, small values of xinc and yinc give the smoothest, though slowest, movement. Finally, altering the colour in which the character is printed after it's started being moved can cause some odd effects due to the XORing together of different colours.

Before leaving this routine, let's take a closer look at the Shape Table. As already said, each entry is 6 bytes long, the first six bytes of the Table being left set to 0. The first six bytes are called Entry 0, the second Entry 1 and so on. The 'char' parameter of the CALL statement is used to indicate which Shape Table entry you wish to use. So, let's say that we want Entry 1 to be a character 254, in Colour 1, starting at 2ØØ,2ØØ on the screen. The entry begins at address (TABLE+6) where TABLE is the start address in memory of the Shape Table. The full entry for this would be:



So, when we want to put this character on the screen for the first time, we'd use a BASIC statement to print CHR$(254) to position 2ØØ,2ØØ in graphics pen 1.

Several Entries can be put into the Shape Table and a subroutine can be written to move all the characters at once, as shown below.

```
4ØØØ  REM Character moving routine
4Ø1Ø  number=6: REM 6 user defined characters
4Ø2Ø  FOR char=1 TO 6
4Ø3Ø  CALL 42ØØØ,char,xinc,yinc
4Ø4Ø  NEXT char
4Ø5Ø  RETURN
```

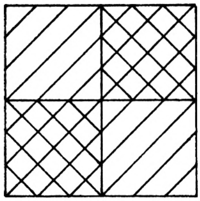
When used in conjunction with the EVERY statement from BASIC, the movement of the characters around the screen can be made to occur at set intervals, the various xinc and yinc values being altered in the main body of the program.

The effects, by the way, of xinc and yinc are quite straightforward. Positive xinc values cause a move to the right, negative xinc values cause a move to the left. Positive yinc values will cause a move up the screen and negative yinc will move down the screen.

We'll now look at a similar routine for moving characters of two colours. The principles that will be discussed can be also applied to characters having more than two colours. The first task here is to see how we can print characters containing two or more colours to the screen. This might well be useful to you in other routines as well.

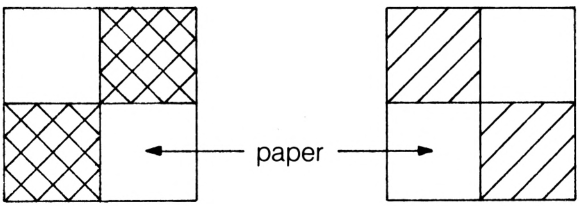
Multicoloured Characters

The answer lies in the use of the XOR graphics mode. Imagine that we want to produce a two coloured character like the one shown below.



Typical Two Colour Character

Two of the squares are of one colour, and two are of another colour. From BASIC, such a character can be put on the screen using TAG, as we'll now see. The first task is to define a user defined character for each foreground colour that the final composite character is to possess. Thus, for a two colour character we will have to define two user defined characters. Note that no areas should overlap. If this happens when we are using XOR, then some rather odd colour effects can occur. Thus for the above character, we might define two characters like:

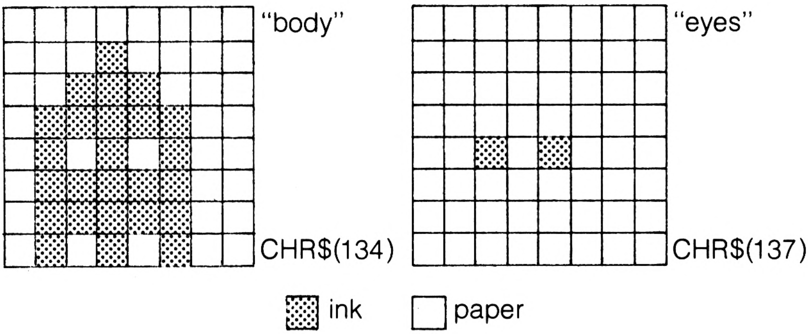


Conveniently, these are available in the Amstrad Character set as CHR$(134) and CHR$(137). All we do now is use TAG to print the character on top of each other at graphics coordinates. This will superimpose the two images. If we print each of them in a different colour, then we'll get our two colour character. The below BASIC program will do this.

```
 100  MODE 1
 110  GOSUB 1000
 120  FOR I=0 TO 300:NEXT
 130  GOSUB 1000
 140  GOTO 120
1000  REM subroutine to print character
1010  TAG
1020  PLOT 1000,1000,1:REM set up colour for first char
1030  MOVE 100,100:REM move to position
1040  PRINT CHR$(134);:REM print the character
1050  TAGOFF:PRINT CHR$(23)+CHR$(1):REM into XOR mode
```

```
1Ø6Ø  TAG
1Ø7Ø  PLOT 1ØØØ,1ØØØ,2:REM set colour for second char.
1Ø8Ø  MOVE 1ØØ,1ØØ:REM move to position
1Ø9Ø  PRINT CHR$(137);:REM print the second char.
11ØØ  RETURN
```

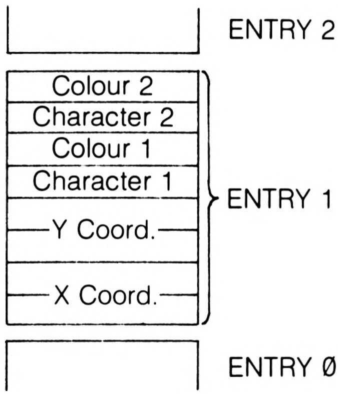
Run the program, and you will see the two coloured shape repeatedly drawn and erased. You might like to examine other characters. If you are interested in games, the below two character grids might be of interest.



The body of this beasty could be printed in yellow, and the eyes in red. Again, remember that there should be no overlap. If there is, some interesting colour effects can be obtained, and you might be interested in trying out some of these effects.

We will now see a program that allows us to move two or more coloured characters around the screen. Although the routine given is for two coloured characters, the following notes should allow you to modify the program if you want to.

MOVECHAR2, as I've rather imaginatively called the program, is very similar to MOVECHAR, as an examination of the listing will show. The difference starts in the Shape Table, where there is now a character and colour entry for each separate 'mask' that goes to make up the composite character. Thus a two colour routine will have two character and colour entries, one for each separate coloured part of the image. Thus for a two colour Character, the Shape Table entries would be of the form:



Two Colour Shape Table Entry

Within the body of the program itself, the main difference is the presence of a second character printing routine at the erasure and redrawing stages to erase and redraw the second coloured character. Obviously, for a three colour character there would have to be a third drawing and erasing operation to deal with the third coloured character. In addition, a three colour character movement routine would need an extra two bytes in the Shape Table Entry to define the third character and colour.

MOVECHAR2

This routine is used to move two coloured characters across the screen in any mode. See the below notes for details of use.

Entry Requirements: From BASIC, CALL address,char,xinc,yinc where char, xinc and yinc have the same significance as in MOVECHAR. From Machine Code routines, A =3 and IX holds the address of a parameter block that is identical in structure to that for MOVECHAR.

Exit Conditions: All registers are corrupt. The graphics pen colour will be that of the last character drawn. The graphics mode will be 'Force' or 'Absolute' mode and the graphics cursor will be at the position to which the character has been moved.

Length: 175 Bytes, not including the Shape Table.

MVCHR2

```
1000  MODE 1
1010  MEMORY 39999
1020  GOSUB 1080
1030  CLS
1040  J=1
1050  PLOT 1000,1000,1 : MOVE 100,100 : PRINT CHR$(23)+CHR$(1
      ) : TAG : PRINT CHR$(134); : PLOT 100,100,2 : PRINT CHR$
      (137); : TAGOFF
1060  FOR I=1 TO 300 : CALL 42000,1,J,0 : CALL &BD19 : : NEXT

1070  END
1080  ASSEMBLE
1090  '          ORG      42000
1100  '          CP       3
1110  '          RET      NZ
1120  '          LD       B,(IX+4)
1130  '          PUSH     IX
1140  '          LD       IX,TABLE        ; start of shapetbl
1150  ' LOOP     INC      IX              ; get entry
1160  '          INC      IX
1170  '          INC      IX
1180  '          INC      IX
1190  '          INC      IX
1200  '          INC      IX
1210  '          INC      IX
1220  '          INC      IX
1230  '          DJNZ     LOOP
1240  '          LD       (TEMP),IX       ; store start entry
1250  '          LD       E,(IX)
1260  '          LD       D,(IX+1)        ; get x coordinate
1270  '          LD       L,(IX+2)        ; get y coordinate
1280  '          LD       H,(IX+3)
1290  '          PUSH     HL
1300  '          PUSH     DE
1310  '          CALL     &BBC0           ; move to the position

1320  '          LD       A,(IX+5)        ; set the first colour

1330  '          CALL     &BBDE
1340  '          LD       A,1             ; set the XOR graphics

1350  '          CALL     &BC59           ; mode
1360  '          LD       A,(IX+4)
1370  '          CALL     &BBFC           ; print first char
1380  '          LD       A,(IX+7)
1390  '          CALL     &BBDE          ; get second colour
1400  '          POP      DE
1410  '          POP      HL
1420  '          PUSH     HL
1430  '          PUSH     DE
1440  '          CALL     &BBC0           ; move to the position

1450  '          LD       A,(IX+6)
1460  '          CALL     &BBFC           ; print 2nd char
1470  '          POP      DE
1480  '          POP      HL
1490  '          POP      IX              ; recover IX
1500  '          PUSH     HL
1510  '          PUSH     DE
1520  '          POP      HL
```

```
1530  '          LD      C,(IX+2)       ; add the increment to
1540  '          LD      B,(IX+3)       ; to x coordinate
1550  '          ADD     HL,BC
1560  '          PUSH    HL
1570  '          POP     DE             ; get it into DE
1580  '          PUSH    IX
1590  '          LD      IX,(TEMP)
1600  '          LD      (IX),E         ; store in table
1610  '          LD      (IX+1),D
1620  '          POP     IX
1630  '          LD      C,(IX+0)       ; add the increment to

1640  '          LD      B,(IX+1)       ; the y coordinate
1650  '          POP     HL
1660  '          ADD     HL,BC
1670  '          LD      IX,(TEMP)
1680  '          LD      (IX+2),L       ; store in table
1690  '          LD      (IX+3),H
1700  '          PUSH    HL
1710  '          PUSH    DE
1720  '          CALL    &BBC0          ; moveto new location
1730  '          LD      A,(IX+5)       ; get first colour
1740  '          CALL    &BBDE
1750  '          LD      A,(IX+4)       ; print first char
1755  '          CALL    &BBFC
1760  '          POP     DE
1770  '          POP     HL
1780  '          CALL    &BBC0
1790  '          LD      A,(IX+7)       ; get second colour
1800  '          CALL    &BBDE
1810  '          LD      A,(IX+6)       ; get second character

1820  '          CALL    &BBFC
1830  '          LD      A,0            ; set 'force' graph
1840  '          CALL    &BC59          ; mode.....
1850  '          RET                    ; and finish.
1860  ' TEMP     WORD    00
1870  ' TABLE    WORD    00             ; shape table
1880  '          WORD    00             ; is a dummy one
1890  '          WORD    00
1900  '          WORD    00
1910  '          WORD    100            ; shape table entry 1
1920  '          WORD    100            ; y coordinate
1930  '          BYTE    134            ; char # 1
1940  '          BYTE    1              ; colour # 1
1950  '          BYTE    137            ; char # 2
1960  '          BYTE    2              ; colour # 2
1970  ' END
1980  RETURN
```

```
FE 03 C0 DD 46 04 DD E5 DD 21 C1 A4 DD 23 DD 23 DD 23 DD
23 DD 23 DD 23 DD 23 DD 23 10 EE DD 22 BF A4 DD 5E 00 DD
56 01 DD 6E 02 DD 66 03 E5 D5 CD C0 BB DD 7E 05 CD DE BB
3E 01 CD 59 BC DD 7E 04 CD FC BB DD 7E 07 CD DE BB D1 E1
E5 D5 CD C0 BB DD 7E 06 CD FC BB D1 E1 DD E1 E5 D5 E1 DD
4E 02 DD 46 03 09 E5 D1 DD E5 DD 2A BF A4 DD 73 00 DD 72
01 DD E1 DD 4E 00 DD 46 01 E1 09 DD 2A BF A4 DD 75 02 DD
74 03 E5 D5 CD C0 BB DD 7E 05 CD DE BB DD 7E 04 CD FC BB
D1 E1 CD C0 BB DD 7E 07 CD DE BB DD 7E 06 CD FC BB 3E 00
CD 59 BC C9 C9 A4 00 00 00 00 00 00 00 00 7A 00 64 00 86
01 89 02
```

Notes With care, the routine can be relocated. It will be necessary to alter the address given in the above bytes, which are intended for address 42ØØØ, for the Shape Table and the variable 'TEMP'. As with MOVECHAR, a character has to be first positioned at it's start position for it's first appearance on the screen using TAG from BASIC, to prevent the 'shadow' of the character being left on the screen as the character moves away from it's start position. Again, POKEs will be needed to set up the Shape Table Entries for each entry. The below BASIC program will demonstrate the machine code of MOVECHAR2. I have assumed that the machine code is at address 42ØØØ, and that the Shape Table entry given above has been included. Note that, as with MOVECHAR, the first entry in the Shape Table, Entry Ø, is left set to all zeros.

```
100 MODE 1
110 xinc=1
120 yinc=1
130 PLOT 1000,1000,1:REM set up colour 1
140 PRINT CHR$(23)+CHR$(1):REM set XOR mode
150 MOVE 100,100: REM move to start position
160 TAG:PRINT CHR$(134);:REM print first character
170 PLOT 1000,1000,2:REM set second colour
180 MOVE 100,100:REM move to start position
190 PRINT CHR$(137);:REM print second character
200 TAGOFF
210 FOR I=0 TO 300
220 CALL 42000,1,xinc,yinc
230 FOR J=0 TO 20
240 NEXT J:REM delay loop
250 NEXT I
260 END
```

Again, should you wish to re-run this routine you will have to POKE the start values back into the Shape Table. In fact, a simple machine code program could be written that takes as parameters all the various pieces of information that are needed for a Shape Table entry and puts them into the correct place. I'll leave that for you to write!

The delay loop at lines 23Ø-24Ø of the above program may need to be altered, depending upon the rest of the program in which the characters are being moved. Due to the addition of more printing routines, the moving of multi-coloured characters is a slower process than the moving of single coloured characters.

As an example of a simple BASIC subroutine to set up a Shape Table Entry for the MOVECHAR2 program, look at the below subroutine. It assumes that the variable 'table' holds the address of the very first byte of the Shape Table.

```
1000  entry=table+(n*8):REM n=Entry Number
1010  POKE entry,100:REM x coordinate
1020  POKE (entry+1),0
1030  POKE (entry+2),100:REM y coordinate
1040  POKE (entry+3),0
1050  POKE (entry+4),134:REM first character
1060  POKE (entry+5),1:REM first colour
1070  POKE (entry+6),137:REM second character
1080  POKE (entry+7),2:REM second colour
1090  RETURN
```

In both MOVECHAR and MOVECHAR2 it's only on the first appearance of the character on the screen that the character to be moved has to be printed to the screen. After this, to move the character to any desired screen location, simply provide the correct xinc and yinc values.

That completes this Chapter of graphics handling routines. We'll now go on to look at keyboard handling from within machine code routines, with some useful routines for string entry, games playing and other applications.

6. Keyboard Operations

In Locomotive BASIC, we're lucky to have a very sophisticated range of commands available to allow us to do such things as setting the repeat rate of individual keys, changing the character returned by keys and so on. In addition, we have the KEY command that allows us to attach strings of characters to given keys. And, of course, we've got the usual INPUT, INKEY and INKEY$ commands. However, in machine code routines we're usually concerned with simply finding out whether a key has been pressed, and this is extremely simple from our own programs due to the provision of excellent Firmware facilities. In this Chapter, I'll give you a variety of routines that will be of use in both machine code and BASIC programs. So, without further ado, here we go.

NORESET

This routine alters the behaviour of the computer on pressing the SHIFT-CTRL-ESC sequence of keys. Instead of totally resetting the machine, it is totally ignored in a running program, and simply generates the *Break* message in command mode. Similarily, ESC is totally disabled in a running program. It thus gives, to a running program, a high degree of protection! The routine is relocatable.

Entry Requirements: From BASIC- CALL address,n where n =Ø disables reset and n =1 causes SHIFT-CTRL-ESC to generate the normal reset.
From machine code, IX points to a single byte holding 'n' and A =1.

Exit Conditions: AF Corrupt.

Length: 22 Bytes.

NRSET

```
1000  MEMORY 39999
1010  GOSUB 2000
1020  CALL 40200,1
1030  END
2000  |ASSEMBLE
2010  '          ORG     40200
2020  '          CP      1
2030  '          RET     NZ
2040  '          LD      A,(IX)
2050  '          CP      0
2060  '          JR      Z,DISABLE
2070  '          LD      A,195
2080  '          LD      (&BDEE),A
2090  '          RET
2100  ' DISABLE  LD      A,201
2110  '          LD      (&BDEE),A
2120  '          RET
2130  ' END
2140  RETURN
```

```
FE 01 C0 DD 7E 00 FE 00 28 06 3E C3 32 EE BD C9 3E C9 32
EE BD C9
```

Notes This routine, as already mentioned, totally disables ESC from a running program, and so if you execute a

CALL 40200,0

command, and then get into a continuous loop...tough! This line should only be executed when you've got a working program. It functions by altering the first byte of one of the jump block entries to hold the code for a RET instruction. When the jump block is entered, after either of the above key sequences is entered, then an immediate return is made. To restore behaviour to normal, the old byte, which is 195, is put back as the first byte in the Jump Block entry.

GET

Most computers have a function called GET, whose role is to cause program execution to cease until a key is pressed. The function then returns, as it's result, the ASCII code of the key that was pressed. Thus

G =GET

will return the ASCII code in the variable G. Amstrad BASIC doesn't possess such a function, and we usually simulate it using a couple of lines of BASIC like:

```
1010  G$=INKEY$: IF G$="" THEN GOTO 1010
1020  G =ASC(G$)
1030  RETURN
```

Here is a machine code routine to do the GET function without the need for the above BASIC lines.

Entry Requirements:	From BASIC, CALL,address,@G% where G% is a previously defined variable in which the ASCII code of the key pressed will be returned. From machine code, it is better just to call the firmware routine directly.
Exit Conditions:	AF, HL Corrupt
Length:	18 Bytes.

```
GET


1000  MEMORY 39999
1010  GOSUB 2000
1020  char%=0
1030  CALL 40200,@char%
1040  PRINT char%
1050  GOTO 1030
2000  |ASSEMBLE
2010  '         ORG       40200
2020  '         CP        1
2030  '         RET       NZ        ; if wrong no. parameters,
2035  '                             ; return
2040  '         LD        L,(IX)
2050  '         LD        H,(IX+1); get address of char%
2060  '         CALL      &BB18
2070  '         LD        (HL),A
2075  '         XOR       A         ; clear A
2076  '         INC       HL
2077  '         LD        (HL),0
2080  '         RET
2090  ' END
2100  RETURN


FE 01 C0 DD 6E 00 DD 66 01 CD 18 BB 77 AF 23 36 00 C9
```

Notes The variable used to hold the returned ASCII code must, as is usual with variables prefixed by '@', have been previously initialised in some way, even if it's just been set to zero. With regard to the Firmware routine, the ASCII code is returned in the A register with the C flag set to 1.

Other keyboard routines are available to the machine code programmer, but they require no setting up and so can be called directly from your machine code routines. They offer no new facilities to the BASIC programmer.

READ KEY

This routine, called at &BB1B, returns a code if a key is being pressed at the instant of the routine being called. It doesn't wait for a key to be pressed. It is thus rather useful in games programs where delays are not needed. If a key was pressed, then on return from the routine the C flag will be set to 1 and A will hold the key code. Otherwise C will be set to Ø.

TEST KEY

This is a bit like the BASIC INKEY(n) function, in that it tests for a certain key being pressed at the instant of the routine being called. On exit, Z=Ø if the key was pressed and Z=1 if the key in question wasn't pressed. The routine is called at address &BB1E, with the A register holding the relevant key number.

It is occasionally useful when programming to get information about the current status of the SHIFT, CTRL, CAPS LOCK and SHIFT LOCK states. This enables you to detect 'odd' key sequences, such as SHIFT-CTRL-ENTER if you were so inclined. The routine is called STATUS.

STATUS

Returns the current status of the Shift, CTRL, Shift Lock and Caps Lock keys. The routine is relocatable.

Entry Requirements: From BASIC, CALL address,@stat%
From machine code, IX points to a parameter block with A=1. The parameter block is a two byte block holding, low byte first, the address of a two byte location where you want the status byte to be returned.

Exit Conditions: All Registers Corrupt.

Length: 31 Bytes.

```
STATUS


1000  MEMORY 39999
1010  GOSUB 2000
1020  char%=0
1030  CALL 40200,@char%
1035  PRINT HEX$(char%)
1036  INPUT a$
1040  GOTO 1030
2000  |ASSEMBLE
2010  '          ORG        40200
2020  '          CP         1
2030  '          RET        NZ
2040  '          LD         L,(IX+0)
2050  '          LD         H,(IX+1)
```

```
2060  '            PUSH     HL
2070  '            CALL     &BB1E       ; get SHIFT/CTRL status
2080  '            PUSH     BC          ; save the BC register
2090  '            CALL     &BB21       ; get CAPS/ S/LOCK stat
2100  '            LD       A,L
2110  '            AND      A,128       ; con CAPS lock stat
2120  '            LD       L,A         ; 128 or 0 and save it
2130  '            LD       A,H         ; con SHIFT lock stat
2140  '            AND      1           ; to 1 or 0 and ......
2150  '            ADD      L           ; add to L to get comp
2160  '            POP      BC          ; value. Now get BC
2180  '            POP      HL
2190  '            LD       (HL),C      ; get SHIFT/CTRL in stat%

2200  '            INC      HL
2210  '            LD       (HL),A      ; get lock stat in stat%
2220  '            RET
2230  ' END
2240  RETURN
```

```
FE 01 C0 DD 6E 00 DD 66 01 E5 CD 1E BB C5 CD 21 BB 7D E6
80 6F 7C E6 01 85 C1 E1 71 23 77 C9
```

Notes The value stored in the status byte locations or the value returned to BASIC in the stat% variable, will need to be decoded before the status of the various keys can be extracted from it. It's best to consider it in a 4 digit hexadecimal format, which you could get in BASIC by using

```
stat%=0:CALL 40200,@stat%:stat$=HEX$(stat%)
PRINT stat$
```

The status can be resolved using the below Table.

	CAPS LOCK	SHIFT LOCK	SHIFT	CTRL
ON	&0100	&8000	&0020	&0080
OFF	&0000	&0000	&0000	&0000

A couple of examples to clarify the use of the Table. If the Shift Lock was on, and CTRL was being pressed at the same time, then a status value of

&8000+&0080
=&8080

Similarily, if a value of &A0 was to be returned, an examination of the Table would reveal that for this value to be obtained from the values in the table, it requires that both the SHIFT ad CTRL keys were pressed. (&20+&80). The routine returns the LOCK statuses that were prevelant at the last occasion that the machine was awaiting input of some kind.

One thing that you have probably noticed during your Amstrad programming is that keys pressed during long programming loops are stored in the keyboard buffer and appear in the next INPUT or INKEY statement to be encountered after the loop has finished. Try the below. Once you've typed 'ENTER', press a couple of other keys, then see how they turn up in the INPUT prompt line.

```
FOR I=Ø TO 3ØØØ:NEXT I:INPUT a$
```

This can cause confusion if you're not expecting it, and if the program is waiting for a key to be pressed before going on then these stored up key presses can cause the program to continue without waiting! Many machines have a means by which such characters can be removed from the keyboard buffer; this process is called FLUSHING the keyboard buffer. It will remove all keys in the buffer at that time, and so if used immediately before a 'GET' style command ensures that the machine doesn't crash on regardless. The routine below performs this task, and is called FLUSH.

FLUSH

Flushes the keyboard buffer.

Entry Requirements: CALL address from both BASIC and Machine Code.

Exit Conditions: AF Corrupt

Length: 6 Bytes.

```
FLUSH


1000  MEMORY 39999
1010  GOSUB 2000
1020  END
2000  |ASSEMBLE
2010  '          ORG     40200
2020  ' LOOP     CALL    &BB09    ; get char from buffer
2030  '          JR      C,LOOP   ; if C=1, more char's ?
2040  '          RET
2050  ' END
2060  RETURN


CD 09 BB 38 FB C9
```

Notes To see the routine in action, try the following. The routine is relocatable, but I've assumed that the bytes are at address 4Ø2ØØ.

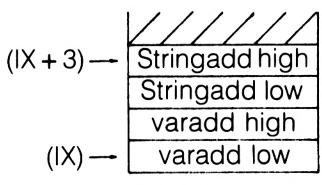
```
1Ø  FOR I=1 TO 3ØØØ:NEXT I
3Ø  INPUT a$
```

Running this, pressing a few keys when the machine is executing line 1Ø, will put some keys into the buffer which will then show up when the INPUT statement is executed. Repeat the run, now, but with a CALL 4Ø2ØØ at line 2Ø. The extra key presses will now be dumped, and the only characters that show up in the INPUT statement will be those entered after line 2Ø has been executed.

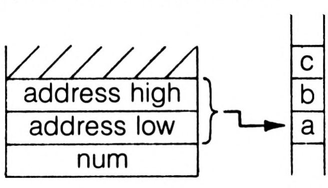
WAITKEY

This routine accepts as its parameters a string of characters and an '@' prefixed variable. The routine then waits until the program detects the pressing of one of the character keys specified in the parameter string. The @ variable then returns the position within the parameter string of the character that has been pressed. The routine is relocatable.

Entry Requirements: From BASIC, CALL address,@a$,@p% where a$ has been previously set up to hold the characters that the routine is to wait for, p% must be set to Ø before being called.
From machine code, things are a little more difficult. IX points to a parameter block and A =2. In the parameter block, stringadd is the address of a Descriptor Block like the one shown. 'varadd' is the address of a single byte location that will hold, after the code has been executed, the position of the key within the data block of acceptable ASCII codes that are pointed to in the Descriptor Block. With regard to the Descriptor Block, num is the number of ASCII codes stored in the table of acceptable ASCII codes pointed to by 'address'.



WAITKEY Parameter Block



WAITKEY Descriptor Block

Exit Conditions: All the registers are corrupt. An immediate exit is made if the wrong number of parameters has been passed over, or if A is not equal to 2.

Length: 63 Bytes.

```
WAITKEY


1000   MEMORY 39999
1010   GOSUB 2000
1020   A$="abcdef" : p%=0
1030   CALL 40200,@A$,@P%
1040   END
2000    |ASSEMBLE
2010   '               ORG         40200
2020   '               CP          2
2030   '               RET         NZ
2040   '               LD          L,(IX)
2050   '               LD          H,(IX+1)
2070   '               PUSH        HL           ; save address of
2075   '                                        ; var. on stack
2080   '               LD          L,(IX+2)     ; get des block add
2090   '               LD          H,(IX+3)
2100   '               LD          A,(HL)       ; no. char in strg
2110   '               INC         HL           ; get add of
2120   '               LD          C,(HL)       ; the string in to the

2130   '               INC         HL           ; BC pair.....
2140   '               LD          B,(HL)
2150   '               PUSH        BC
2160   '               POP         HL           ; and then into HL
2170   '               LD          (TEMP),A     ; save len/strg to sea
       rch
2171   '               LD          (TEMP2),HL   ; save add of strg
2180   '   LOOP2       LD          A,(TEMP)     ; get leng. into B
2190   '               LD          B,A
2195   '               LD          HL,(TEMP2)   ; recover address
2200   '               PUSH        BC
2210   '               PUSH        HL
2220   '               CALL        &BB18        ; wait for character
2230   '               POP         HL
2240   '               POP         BC
2245   '               LD          E,1          ; initialise a counter

2250   '   LOOP        CP          (HL)         ; comp char with
2260   '               JR          Z,FOUND      ; table, jump if ok
2265   '               INC         E            ; bump up counter..
2270   '               INC         HL           ; look next char
2280   '               DJNZ        LOOP         ; until chars done
2290   '               JR          LOOP2        ; if not found again
2300   '   FOUND       POP         HL           ; recover var add
2310   '               LD          (HL),E       ; put the value in it

2320   '               INC         HL
2330   '               LD          (HL),0
2340   '               RET                      ; back to BASIC
2345   '   TEMP        BYTE        0
2346   '   TEMP2       WORD        00
```

```
2350  ' END
2360  RETURN
```

```
FE 02 C0 DD 6E 00 DD 66 01 E5 DD 6E 02 DD 66 03 7E 23 4E
23 46 C5 E1 32 44 9D 22 45 9D 3A 44 9D 47 2A 45 9D C5 E5
CD 18 BB E1 C1 1E 01 BE 28 06 1C 23 10 F9 18 E7 E1 73 23
36 00 C9 06 91 01
```

Notes The routine is very useful in many applications. For example, vetting a user's response to a question that requires just a 'yes/no' answer. The only response keys of interest are going to be 'Y', 'N', 'n' and 'y'. So, from BASIC we can simply execute the line:

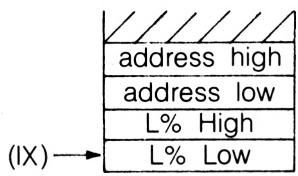
A$="YyNn":p%=Ø:CALL 40200,@A$,@p%

The routine will only return to BASIC when a suitable response has been received from the keyboard, in this case Y,y,N or n. p% will then hold a value that corresponds to the key pressed. Thus if 'Y' was pressed, p%=1. Or, if 'n' was pressed, p%=4.

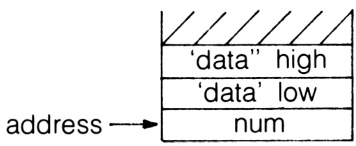
The next routine that we'll look at is a simple 'input' routine that allows the user to type in a string of characters, terminated by the ENTER key. The routine is called INSTRING.

INSTRING

Accepts a string of characters from the keyboard, and stores them in either a string variable or a block of memory.



INSTRING Parameter Block



INSTRING Descriptor Block

Entry Requirements: From BASIC, CALL address, @A$,L% where L% is the maximum number of characters that the user is to be allowed to type in. A$ is a string variable that has been previously initialised to be longer than L%.
From machine code, IX points to a parameter block and A=2. In the parameter block, 'address' is the address of a Descriptor Block.
In the Descriptor Block, num is the number of

bytes that have been set aside for the inputted string. It should be greater than L%. 'data' is the address in memory where the inputted string is to reside.

Exit Conditions: All registers corrupt. The routine is terminated when:
(i) ENTER is pressed.
(ii) The user types in more characters than L%. The string returned in this case will be the first L% characters.
In both cases, a short 'beep' will be generated. The entered characters can then be accessed either in a string variable or in the 'data' area of memory.

Length: 137 Bytes.

INSTRING

```
1000  MEMORY 39999
1010  GOSUB 1070
1020  a$="JOE SOAP   G                   "
1030  CALL 41000,@a$,10
1040   PRINT
1050  PRINT a$
1060  END
1070   ASSEMBLE
1080  '            ORG      41000
1090  '            CP       2
1100  '            RET      NZ
1110  '            LD       A,(IX)
1120  '            LD       L,(IX+2)
1130  '            LD       H,(IX+3)
1140  '            CP       (HL)
1150  '            LD       A,(HL)
1160  '            LD       (TEMP),A
1170  '            JR       C,OK         ; only if length of string
1180  '                                  ; is longer than specified
1190  '                                  ; len do we go on
1200  '            RET
1210  ' OK         INC      HL
1220  '            LD       C,(HL)
1230  '            INC      HL
1240  '            LD       B,(HL)
1250  '            PUSH     BC           ; get address of string
1260  '            LD       A,(IX)
1270  '            LD       B,A          ; proposed length
1280  '            POP      IX
1290  '            LD       (TEMPIX),IX
1300  '            PUSH     IX
1310  '            PUSH     BC
1311  '            LD       A,(TEMP)
1312  '            LD       B,A
1320  ' CLEAR      LD       (IX),32
1330  '            INC      IX
```

```
1340  '          DJNZ     CLEAR     ; clear string to spaces
1350  '          POP      BC
1360  '          POP      IX
1370  ' LOOP     PUSH     BC        ; wait for a char.
1380  '          CALL     &BB18
1390  '          CP       127
1400  '          JR       Z,DELETE
1410  '          CP       13
1420  '          JR       Z,FINISH ; if enter, quit
1430  '          LD       (IX),A
1440  '          CALL     &BB5A     ; print it to screen
1450  '          INC      IX        ; next location to fill
1460  '          POP      BC        ; now see if max. len
1465  '                             ; exceeded
1470  ' LOOP1    DJNZ     LOOP
1480  '          JR        FINISH2 ; jump past POP
1490  ' FINISH   POP       BC       ; clear up stack
1500  ' FINISH2 LD        A,7
1510  '          CALL     &BB5A     ; beep for fun!!
1520  '          RET
1530  ' DELETE   LD       HL,(TEMPIX) ; here if del pressed
1540  '          XOR      A
1550  '          PUSH     IX
1560  '          POP      DE
1570  '          SBC      HL,DE        ; if nothing in string...
1580  '          JR       NZ,DOIT
1590  '          LD       A,7
1600  '          CALL     &BB5A
1610  '          JR       NODO         ; beep and exit to NODO
1620  ' DOIT     LD       A,8
1630  '          CALL     &BB5A
1640  '          LD       A,32
1650  '          CALL     &BB5A
1660  '          LD       A,8
1670  '          CALL     &BB5A        ; move back, print space
1680  '          POP      BC           ; fudge character counter

1690  '          INC      B
1700  '          INC      B
1710  '          DEC      IX           ; fudge string position
1720  '          LD       (IX),32      ; set to space
1730  '          JR       LOOP1        ; back
1740  ' NODO     POP      BC
1750  '          INC      B
1760  '          INC      B
1770  '          JR       LOOP1        ; back to main loop
1780  ' TEMPIX   WORD     00
1785  ' TEMP     BYTE     0
1790  ' END
1800  RETURN
```

```
FE 02 C0 DD 7E 00 DD 6E 02 DD 66 03 BE 7E 32 B0 A0 38 01
C9 23 4E 23 46 C5 DD 7E 00 47 DD E1 DD 22 AE A0 DD E5 C5
3A B0 A0 47 DD 36 00 20 DD 23 10 F8 C1 DD E1 C5 CD 18 BB
FE 7F 28 18 FE 0D 28 0D DD 77 00 CD 5A BB DD 23 C1 10 E9
18 01 C1 3E 07 CD 5A BB C9 2A AE A0 AF DD E5 D1 ED 52 20
07 3E 07 CD 5A BB 18 1A 3E 08 CD 5A BB 3E 20 CD 5A BB 3E
08 CD 5A BB C1 04 04 DD 2B DD 36 00 20 18 C9 C1 04 04 18
C4 91 01 1A
```

Notes The routine is immediately exited if the number of parameters isn't correct, or if A does not hold the value 2. Also, if the value of L% is greater than the length of the string (from BASIC) or the amount of 'data' space available (from machine code), then an immediate return is made. Otherwise, once the routine is called you can type in characters until you press ENTER or the number of characters typed in exceeds L%. The characters will be printed to the screen as well as being put in the string area. Delete will work, and the routine will NOT allow you to delete past the beginning of the string! It will 'beep' if you try this.

Try the below demonstration routine in BASIC. The bytes above are for address 41ØØØ, and I've assumed that the machine code is at that address.

```
100  MODE 2
110  P% =1Ø: REM number of characters.
120  A$="            ":REM 12 spaces in this line
130  CALL 41ØØØ,@A$,P%
140  PRINT:PRINT
150  PRINT A$
160  END
```

If you run this, enter some characters and then list the program you'll notice that the string definition in line 12Ø has been modified to hold the characters that have been typed in. More of this in Chapter 9, when we'll discuss the structure of the Amstrad BASIC program in more detail.

Although ROM routines exist in the Amstrad to do such things as alter the key repeat rates, etc. I do not intend looking at any routines here. It's usually easier to use the BASIC equivalents. So, that is where we'll end our look at keyboard routines.

7. Sound Routines

The Amstrad 464 is capable of some very impressive sound effects as you will no doubt be aware if you've tried to use the extensive sound facilities of the machine from BASIC. Many of these facilities are also available from machine code via the use of Firmware routines. These allow the machine code programmer access to queueing, Tone Envelopes, Amplitude Envelopes, etc. They are documented thoroughly in the Amsoft "Firmware Technical Manual", and there's no point in repeating such information here.

Instead, I want to look at the programming of the Programmable Sound Generator chip directly. Why bother, I hear you say, if we can get effects via the firmware routines? Well, the use of the Firmware often requires that complicated tables of data are set up in memory before the routines are called. The setting up procedures are not so long winded when we use the PSG directly. Although the PSG cannot do everything that is available from BASIC, it is capable of generating tones, noise and has a few amplitude envelopes available to it as well, so it is quite versatile. Tone Envelopes, etc, that are available from BASIC, are produced by the PSG with a little help from the CPU. The advantage about using the PSG directly is that for the sound effects often desired in games programs, etc. it's more convenient to access the PSG directly than to set up all the various tables, etc. that are required for the Firmware routines. Of course, some sound effects will require the use of Firmware routines, but you will find it surprising how much you can get out of the PSG on its own.

Beep!

The simplest sound effect to get is that generated by the below piece of machine code:

```
LD      a,7
CALL    &BB5A
RET
```

This 'beep' is useful for indicating that something has happened or that an error has occurred and so on.

The Programmable Sound Generator

The PSG is a very versatile device, and is relatively simple to use even if it looks a little daunting at first glance. It is a three voice device; that is, it is capable of generating three separate noises at once. It also has a few built in hardware amplitude envelopes, and control of the volume of the sound produced on each channel is controllable. If you are unsure about the meaning of the phrase Envelope, then I suggest that you consult the Amstrad Manual and read Chapter 6. The pitch and amplitude of tones played are all individually controllable. The chip is also capable of producing white noise. All this is done by a collection of 15 registers within the PSG, which can be written to and read from in a similar fashion to CPU registers. The PSG is also capable of performing Input/Output operations, but we won't go into that here. Although the Amstrad OS accesses the chip via the PPI chip, we can't just write directly to PPI registers. Well, we could but it wouldn't be very advisable, due to the complexity of I/O operations on the Amstrad and the possibility that we might mess something up doing this. Instead, Amstrad have supplied us with a useful routine that enables us to write a value to a particular PSG register without any danger of messing things up.

MC SOUND REGISTER

This routine is called at address &BD34, with the C register holding a value between Ø and 255 and the A register holding the number of the PSG register to which you want to write the value. On exit, both AF and BC are corrupt. The below routine, REGISTER, can be used from BASIC to write values to the PSG registers.

REGISTER

A routine to allow the programmer to directly access PSG registers from BASIC

Entry Requirements: From BASIC, CALL address, reg, value where reg =register number and value is the value to be written to the register. From machine code, the ROM routines can be called directly.

Exit Conditions: Not Applicable.

Length: 12 Bytes.

REGISTER

```
1ØØØ  MEMORY 39999
1Ø1Ø  GOSUB 1Ø3Ø
1Ø2Ø  END
1Ø3Ø   |ASSEMBLE
1Ø4Ø  '        org      4Ø2ØØ
1Ø5Ø  '        LD       A,(IX+2)
1Ø6Ø  '        LD       C,(IX+Ø)
1Ø7Ø  '        CALL     &BD34
1Ø8Ø  '        RET
1Ø9Ø  ' END
11ØØ  RETURN
```

```
DD 7E Ø2 DD 4E ØØ CD 34 BD C9
```

Notes The main thing about this routine is that before you can use it, you need some knowledge about the Programmable Sound Generator registers. We'll now have a look at these.

PSG Registers

This section is something of a 'crash course' in PSG registers, and if you really want to get fully versed in the device then the Data Sheet for the device is the AY-3-8912 Data Sheet from General Instruments. However, for programming purposes the data to be presented here will probably be quite adequate. The registers of the PSG are numbered, rather imaginatively, Ø to 15. Registers 14 and 15 are not used for sound generation purposes, but are involved with the I/O side of the PSG. As with all such registers, if you do not know exactly what you're doing, DO NOT TOUCH!

Registers Ø to 5

These control the pitch of the note played on Channels 1 to 3. The pitch of Channel 1 is controlled by Registers Ø and 1, that of Channel 2 by Registers 2 and 3 and that of Channel 3 of Registers 4 and 5.

The registers are effectively 12 bit registers; the lower 8 bits are in the lower numbered register of each pair and the higher 4 bits are held in the higher numbered register of each pair. The lower 8 bit register, i.e. Register Ø for Channel 1, is called the Fine Tune Control Register and the higher 4 bit register is called the Coarse Tune Control Register. The reason for this is obvious; a single count in the Coarse Control Register has quite an effect on the pitch of the tone being played, while a single count in the Fine Tune register gives a change in pitch that is barely discernable. Thus Register Ø is the Fine Tune Control Register for Channel 1 and Register 1 is the Coarse Tune Control Register for the same Channel.

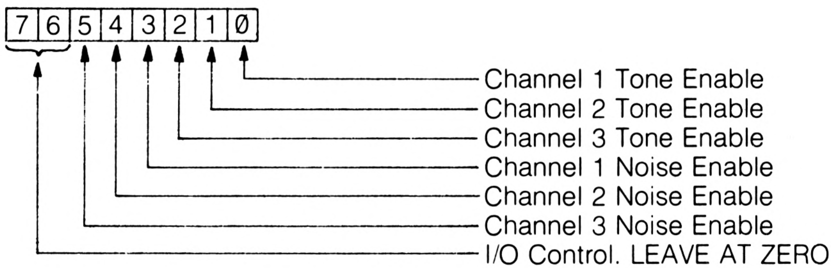
The larger the number placed in these registers, the lower is the pitch of the tone generated on that particular channel.

Registers 8 to 10

These are called the Amplitude Control Registers of the PSG. There is one for each Channel, and they are all 5 bit registers. That is, they can hold a value between Ø and 31. However, if Bit 4 is set to 1, the Channel in question behaves in a special way, as we'll soon see. Normally, the volume of sound produced on each Channel depends upon the value in Bits Ø to 3. A value of Ø gives silence, and a value of 15 gives the loudest sound. If Bit 4 is set to 1, then the amplitude of the sound generated on that Channel is under the control of one of the PSG's built in Amplitude Envelopes rather than under the control of the lower 4 bits of the appropriate Amplitude Control Register. We'll examine these Amplitude Envelopes in greater detail later. Register 8 controls the volume of Channel 1, Register 9 controls Channel 2 and Register 1Ø controls the amplitude of Channel 3.

It is not enough on the PSG to set up the amplitude and pitch for a particular channei and expect the PSG to play the note selected. Whether a sound is generated on a particular channel or not depends upon the status of certain bits in Register 7, which is the Control Register of the PSG.

Register 7



Bits 6 and 7 of this register are concerned with the control of the Input/Output facilities of the PSG. In the Amstrad, these would appear to be involved with the Keyboard in some way, 'cos if you mess about with Bit 6 and set it to 1 the keyboard appears to go dead. The only way out, from direct mode, is to turn off! Of course, in a program you could have other instructions to reset this bit, but I suggest that it is best to leave both bits 6 and 7 set permanently to Ø.

Bits Ø to 2

These control the generation of tone on Channels 1 to 3. If a bit is set to Ø, then tone will be generated on that channel, assuming a suitable value has been put in the Amplitude Control Register for that Channel. Thus to play a Tone on Channel 1, the following steps would have to be gone through.

(i) A suitable pitch would have to be in Registers Ø and 1.

(ii) A suitable Amplitude would have to be in Register 8.

(iii) Register 7 would have to be set to &XØØ111110.

When a bit for a given Channel is set to Ø in this way, the Channel concerned is said to be ENABLED. If the bit in question is set to 1, no tone will be played and the Channel is said to be DISABLED for tone production.

Setting all three of these bits to zero would therefore allow tones to be played simultaneously on all three channels.

Bits 3 to 5

These are called the Noise Enable Bits and are responsible for controlling whether or not white noise is played on any or all of the three channels. More of noise in a while. Again, setting one of these bits to Ø will cause white noise to be played on a particular channel, at the volume set by the Amplitude Control Register (R.8 to R.1Ø) for that Channel. Setting a bit to 1 disables noise on that Channel.

It is possible to have both Tone and Noise playing at the same time on the same Channel, by setting both the Noise and Tone enable bits for that Channel to Ø. However, there is no separate Noise Amplitude Control Register, and the Noise and Tone produced on a particular Channel simultaneously are played at the same volume. Noise can also, of course, be played under Envelope Control on a particular Channel in a similar fashion to Tone.

Noise Generation

The Programmable Sound Generator is capable of producing white noise over a range of 'frequencies', if such a term can be applied to noise. A low pitched noise is a 'rushing' sound, while the higher pitched noise has more of a 'hissy' quality. The pitch of the noise played on all Channels is controlled by a single register.

Register 6

This is the Noise Pitch Control Register. It is a 6 bit register, thus giving noise pitch values between Ø and 31. Ø will give the highest pitched noise and 31 will give the lowest pitched noise. As already mentioned, there is only one Noise Pitch Control Register for all three Channels, so the pitch of noise played on each channel cannot be individually controlled.

Finally, we consider the resident Envelopes of the PSG. These are all Amplitude Envelopes, and can be rather useful. The remaining PSG registers are all concerned with the I/O and Envelope Control.

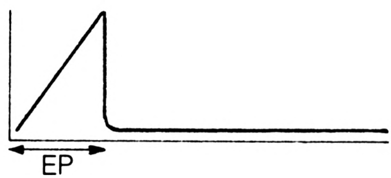
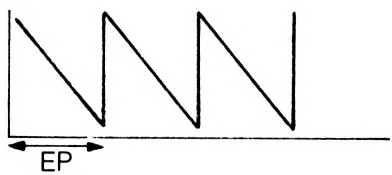
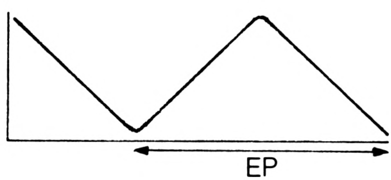
Envelopes

There are two parameters that describe the Amplitude Envelopes that are provided by the PSG.

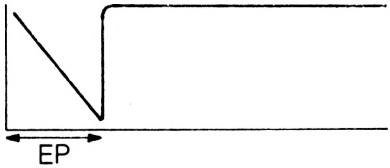
These are the Envelope Number, which describes the 'Shape' of the Envelope, and the Envelope Period, which is a measure of the amount of time that is taken for an Enveloped sound to be played.

Register 13

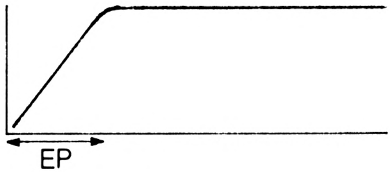
The contents of this register specify the Shape of the Envelope that will be applied to any Enveloped sounds played.

Register 13 Contents	Envelope Shape
1	
4	
8	
10	

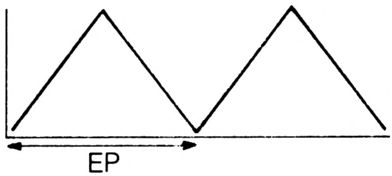
11



13



14



EP stands for Envelope Period, and it's duration is controlled by the value held in the two Envelope Period Registers, R11 and R12. The registers together form a 16 bit value, of which R11 is the low byte and R12 is the high byte. The higher the value in this register, the longer the Envelope Period. The longer the Envelope Period is, the slower is the rate of change of amplitude during the playing of the Envelope.

Whether an Envelope is applied to a tone being played on a particular Channel, or a noise for that matter, depends upon the setting of Bit 4 of that Channel's Amplitude Control Register. ALL the channels that have this bit set will play their tones or noise with the same envelope. This is because of the fact that there is only one Envelope Shape Register and Envelope Period Register.

Sound Techniques

The basic techniques of using the PSG directly are easy to master. Generally, though, you will need to do a little trial and error before good sound effects start coming out of the Amstrad loudspeaker. Although you can turn a Channel off altogether by setting the appropriate Amplitude Control Register to zero. This will work, but it's probably not the best way of doing things, if only because doing it like this kills both noise and tone at the same time. On the whole, it is better to use Register 7 to control the playing of sound. All the other registers for each channel can be set up with, say, Register 7 set to hold &XØØ111111, thus disabling both noise and tone on all channels. Then, when you want to produce the sound, the relevant bits of Register 7 can be set to Ø to enable either noise, tone or both on the desired channels.

Once a sound is being generated, you can alter the contents of the registers. Thus a sound can be gradually faded out under CPU control, or its pitch can be altered. This is how, in fact, the Amstrad Operating System generates its amplitude and pitch envelopes, altering the PSG registers under interrupt control. It should be noted, however, that once started a Channel will continue playing until Register 7 is set up to stop it playing. This is very useful, especially with the PSG's Envelopes, as it means that the CPU can start the PSG off and then go and do something more important.

I'd like to finish this Chapter with a few examples of sound effects which are produced with the PSG's own Envelopes, just to show what is available. Should you wish to produce more exciting effects, loops can be produced in simple machine code routines to update the PSG registers while the sound is being produced. However, these routines should give you something to go on with. All that is necessary to hear them is that the relevant PSG registers be set up with the values given, Register 7 being set up last.

Explosion

Register	Value
6	3Ø
7	&XØØ11Ø111
8	16
11	255
12	3Ø
13	1

Steam Train

Register	Value
6	3Ø
7	&XØØ11Ø111
8	16
11	255
12	Ø
13	14

You might like to try altering the noise pitch by changing the contents of Register 6.

Laser Beam

Register	Value
Ø	255
1	Ø
7	&XØØ11111Ø
8	16
11	1ØØ
12	Ø
13	1Ø

Boing

Register	Value
Ø	Ø
1	9
7	&XØØ11111Ø
8	16
11	191
12	1Ø
13	1

Gunshot

Register	Value
6	3Ø
7	&XØØ11Ø111
8	16
11	255
12	2
13	1

In all these routines, it is assumed that PSG registers not specifically mentioned are set to zero.

8. Cassette Handling Routines

There are a variety of routines in the Amstrad that allow the machine code programmer easy access to the routines that are concerned with cassette tape operations. With all the facilities that are available from BASIC you might ask 'Why Bother' with accessing such routines from machine code programs. Well, if you're writing utility programs that may need to write data to tape, it's useful. Also, a little knowledge of the cassette routines will allow you to produce individualized cassette formats for software protection or for special data files that only particular programs can access.

However, we'll start with a couple of useful Firmware calls that you can use with no setting up.

Motor Control

One vaguely irritating feature about the Amstrad Cassette system is the way in which the PLAY key is disabled except during input, output or catalogue operations. This means that to find a blank piece of tape it's necessary to CAT the tape, thus preventing you from doing anything else while the tape plays.

Two ROM routines can be used to control the motor; CALL &BC6E will turn the motor ON and CALL &BC71 will turn the cassette motor off. After you turn the motor on, there will be a slight pause before the 'Ready' prompt returns. Don't panic! This is just the operating system ensuring that the tape has reached a smooth running speed before returning.

The routine called at address &BC9B is also useful, as it does the machine code equivalent of a CAT command from BASIC. If you use this latter call, DE must point to a 2048 byte of memory that the Firmware routine can use as workspace.

CAT

Not to be confused with the BASIC command of the same name, this machine code routine will load in the header of a tape file. Once in memory, the various details about the file, such as its length, start address etc., are available to use.

Entry Requirements: From BASIC or machine code, simply CALL the routine. There must be a buffer area available. (See Notes.)

Exit Conditions: All Registers Corrupt.

Length: 12 Bytes + 64 Bytes for buffer.

```
DCAT


1000  MEMORY 39999
1010  GOSUB 2000
1020  CALL 40200
1025  name$="" : buffer=40000
1030  FOR I=buffer TO buffer+15 : name$=name$+CHR$(PEEK(I)) :
       NEXT
1040  start=PEEK(buffer+21)+256*(PEEK(buffer+22))
1050  length=PEEK(buffer+24)+256*(PEEK(buffer+25))
1060  PRINT name$ : PRINT "Length :          ";length : PRINT
      "Start Address :   ";start
1065  PRINT : PRINT
1070  GOTO 1020
2000   ASSEMBLE
2010  '         ORG     40200
2020  '         LD      A,&2C        ; sync code expected
2030  '         LD      HL,40000     ; 'buffer' at 40000
2040  '         LD      DE,64        ; no. of bytes to load
2050  '         CALL    &BCA1        ; load them
2060  '         RET                  ; finished
2070  ' END
2080  RETURN


3E 2C 21 40 9C 11 40 00 CD A1 BC C9
```

Notes This routine is relocatable, provided that the Buffer address is altered if this becomes necessary. In this routine, HL is used to pass the buffer address over to the Firmware routine. In the routine above, 40000 has been used as the buffer address. DE holds the number of bytes that the Firmware routine is to load in, which is always 64 for a header.

The A register must hold a value called the 'sync byte' on entry to the Firmware routine, which effectively differentiates between the Header of a file and a Data block of a file. The header for a data block is &16 and that for a header block is &2C. If the sync byte of a block of information on tape does not correspond to the sync byte of the A register, then that information will be ignored.

After the Firmware routine has pulled in the header, we simply return to BASIC. However, it is also possible that the return to BASIC was caused by an error condition of some type. We've made no use of this information here, but you might like to extend the above routine by adding error messages, so here we go.

If the load operation was successful, then the Carry Flag will be set to 1. Otherwise, C =Ø. In this case, the A register holds an error number.

A =Ø ESC was pressed to terminate the operation.

A =1 Overrun error. This occurs if more data was available from tape than was specified in the DE register when the routine at &BCA1 was called.

A =2 This indicates a CRC error. The Cyclic Redundancy Check is the system by which the Operating System can determine if an error has been made in the actual reading of a byte of data from tape. If this occurs, it generally indicates that one or more of the bytes read in from tape are corrupt.

Once we've got the header block in to memory, there remains the job of decoding it. In this routine, we only want the File Name, Total File Length and start address of the file. The remaining bytes of the header we need not worry about here. If you are curious, however, the Firmware Manual will give you the details that you require, as will "The Ins and Outs of the Amstrad" also published by Melbourne House.

File Name This is stored in the first 16 bytes of the header, and this is why a file name for cassette files can only be 16 characters long. If the name is shorter than 16 letters, then it is made up to 16 characters by filling it out with null codes (CHR$(Ø)).

Start Address This is stored in bytes 21 and 22 of the Header Block. The block is numbered from Ø upwards. This address is that from which the data was saved, and is stored in the usual z-8Ø format with the low byte first. There is one point to remember, however. It is only the start address of that particular block of data, and so will be the start address of the whole file only for the very first block of data. For subsequent blocks it is the start address for that particular data block. The block number, which might be useful in these cases, is stored in Byte 16 of the header.

File Length This is the total length of the file saved. It is stored in bytes 24 and 25 of the Header, again with the low byte stored first.

The below BASIC program uses the CAT routine to provide a more detailed catalogue based on the header entries mentioned above. I have found it particularly useful in keeping an eye on my machine code programs, which I tend to keep stored as byte files. By the way, this routine is not designed to help you gain access to other people's programs. It's illegal, so don't do it. I've assumed that the machine code part of the program is at address 40200 and that the buffer for inputted data is at address 40000. The program, when run, waits for headers and prints a few details. Ecape will finish the program.

```
100  buffer=40000
110  CALL 40200
120  name$=""
130  FOR I=buffer TO buffer+15
140  name$=name$+CHR$(PEEK(I))
150  NEXT I
160  start=PEEK(buffer+21)+256*PEEK(buffer+22)
170  length=PEEK(buffer+24)+256*PEEK(buffer+25)
180  PRINT name$
190  PRINT "Length:" ; length
200  PRINT "Start Address:" ; Start
210  PRINT:PRINT
220  GOTO 110
```

Once you have details about the rest of the header, you can easily expand this routine. However, this is adequate for many applications.

CWRITE

This is a routine to write a named file of data to tape. It is effectively a machine code version of the SAVE command when applied to binary files, and for this reason no BASIC entry requirements will be given. I will also give a couple of read routines, thus allowing you to save and load data files from machine code.

Entry Conditions: IX points to a 64 byte header block, as shown in the Notes for this routine.

Exit Conditions: All Registers Corrupt.

Length: 37 Bytes.

```
CWRITE


1000   MEMORY 39999
1010   GOSUB 2000
```

```
1020  CALL 40200
1030  END
2000   ASSEMBLE
2010  '                 ORG      40200
2020  '                 LD       IX,HEADER
2030  '                 PUSH     IX
2040  '                 LD       HL,HEADER
2050  '                 LD       DE,64
2060  '                 LD       A,&2C
2070  '                 CALL     &BC9E
2080  '                 POP      IX
2090  '                 LD       L,(IX+21)
2100  '                 LD       H,(IX+22)
2110  '                 LD       E,(IX+19)
2120  '                 LD       D,(IX+20)
2130  '                 LD       A,&16
2140  '                 CALL     &BC9E
2150  '                 RET
2160  ' HEADER          TEXT     "TEST",0,0,0,0,0,0,0,0,0,0,0,0
2170  '                 BYTE     1,0,0
2180  '                 WORD     1000
2190  '                 WORD     361
2200  '                 BYTE     00
2210  ' END
2220  RETURN
```

```
DD 21 2D 9D DD E5 21 2D 9D 11 40 00 3E 2C CD 9E BC DD E1
DD 6E 15 DD 66 16 DD 5E 13 DD 56 14 3E 16 CD 9E BC C9 54
45 53 54 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 E8
03 69 01 00
```

Notes The header block mentioned is a simplified version of that used by the OS when it carries out BASIC save and load operations. It is important to note that although files saved by CWRITE will show up on BASIC CAT commands, they will not load using the BASIC LOAD command. The file saved by CWRITE consists of a header, then a block of data. The structure of a header block should be of the below form:

Bytes Ø to 15	FILENAME+CHR$(Ø)'s to fill up
Bytes 19 and 20	FILE LENGTH, low byte first
Bytes 21 and 22	START address, low byte first
Bytes 24 and 25	FILE LENGTH, low byte first

All the rest of the bytes in the header should be set to zero. As a more specific example, consider the below Header block, which will save to tape a block of data 1ØØØ bytes long, starting at address 361, with the file name TEST.

```
HEADER   TEXT   "TEST",Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
         BYTE   Ø
         BYTE   Ø
         BYTE   Ø
```

```
WORD  1000
WORD  361
WORD  00
WORD  1000
```

All the rest of the block is set to zero. Note that once the CWRITE routine is called, the file will be saved immediately to tape, without the usual prompts. These could easily be added, if it was important that for a particular application prompts should be added. The only new ROM routine to be used here is CAS WRITE, called at &BC9E. On entry, HL holds the address of the data to be written, DE the length and A the sync byte. We use it twice, once to put the header on tape and a second time to put the data on tape.

Once we've written a file to tape, it's useful to be able to read it back. I offer two routines for this, CREAD and, very imaginatively, CREAD2. The latter is just a more user friendly version of CREAD.

CREAD

This routine reads from tape files written by CWRITE.

Entry Requirements: See Notes.

Exit Conditions: All Registers Corrupt.

Length: 54 Bytes, excluding NAME and HEADER tables.

```
CREAD


1000  MEMORY 39999
1010  GOSUB 2000
1030  END
2000  |ASSEMBLE
2010  '            ORG     40200
2020  ' LOAD       LD      DE,64       ;No bytes in header
2030  '            LD      HL,HEADER   ; put them here
2040  '            LD      A,&2C       ; correct sync byte
2050  '            CALL    &BCA1       ; load header
2060  '            LD      HL,HEADER   ; DE and HL to point
2070  '            LD      DE,NAME     ; file name and
2075  '                                ; desired name
2080  ' LOOP       LD      A,(DE)      ; check each character
2090  '            CP      (HL)
2100  '            JR      NZ,LOAD     ; if not same, load
2105  '                                ; next header
2110  '            INC     HL
2120  '            INC     DE
2130  '            LD      A,(DE)
2140  '            CP      0           ; zero indicates the
2145  '                                ; end of name
2150  '            JR      NZ,LOOP     ; if not end, next char

2160  '            LD      IX,NAME     ; pick up load address
```

```
2170  '         LD      L,(IX+16)
2180  '         LD      H,(IX+17)
2190  '         LD      IX,HEADER    ; pick up length from
2195  '                              ; header
2200  '         LD      E,(IX+24)
2210  '         LD      D,(IX+25)
2220  '         LD      A,&16        ; sync byte
2230  '         CALL    &BCA1
2240  '         RET
2250  ' NAME    TEXT    "TEST",0,0,0,0,0,0,0,0,0,0,0,0
2260  '         WORD    40000        ; load add
2270  ' HEADER  RMEM    64           ; 64 zeros
2280  ' END
2290 RETURN
```

```
11 40 00 21 50 9D 3E 2C CD A1 BC 21 50 9D 11 3E 9D 1A BE
20 EB 23 13 1A FE 00 20 F5 DD 21 3E 9D DD 6E 10 DD 66 11
DD 21 50 9D DD 5E 18 DD 56 19 3E 16 CD A1 BC C9 54 45 53
54 00 00 00 00 00 00 00 00 00 00 00 00 40 9C 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00
```

Notes This routine requires the presence of two tables of data, which I've called NAME and HEADER. The HEADER table is simply a block of 64 bytes into which a header can be loaded. NAME, however, must be set up by the programmer. As its name suggests, it holds the name of the file that you want to read in. It also holds the address to which the file is to be loaded. It is 18 bytes long.

Bytes 0 to 15 This holds the file name, filled out to 16 bytes long with CHR$(Ø)'s.

Bytes 16 and 17 The load address of the file is stored in these two locations, low byte first.

Due to its simplicity, there are some disadvantages associated with this routine. These are mainly the inability to 'break out' of a search for a file name, the lack of messages while the searching is going on and the failure to check on file length. Let's look at the last one first.

The number of bytes loaded by this routine is taken from the header whose name matches the file name of interest. A potential problem exists in that the bytes loaded into the space designated as starting at the load address may overwrite programs or other data that they are not supposed to. Of course, this shouldn't happen if you are careful in program design, but it could. If you are concerned about this happening, then the following steps can be taken.

(i) Designate a particular address in RAM as being the end of the area in to which data read from tape can be loaded.

(ii) When a file header is read in, add its length to that held in the 'load address' entry in the NAME table. If the result exceeds the address mentioned in (i), then do not load it.

CREAD2 does not include such a feature, but you should be able to add it easily enough using the data given here.

CREAD2

A modified version of CREAD.

Entry Requirements: See Notes.

Exit Conditions: All Registers Corrupt.

Length: 215 Bytes, excluding NAME and HEADER.

```
CREAD2

1000  MEMORY 39999
1010  GOSUB 1030
1020  END
1030  |ASSEMBLE
1040  '              ORG       40200
1050  '              LD        IX,SEARCH
1060  '              CALL      SPRINT
1070  ' LOAD         LD        IX,OK
1080  '              CALL      SPRINT
1090  '              CALL      &BB18
1100  '              CP        121
1110  '              JR        Z,YES
1120  '              CP        89
1130  '              RET       NZ
1140  ' YES          LD        DE,64       ;numb bytes in header
1150  '              LD        HL,HEADER   ; put them here
1160  '              LD        A,&2C       ; correct sync byte
1170  '              CALL      &BCA1       ; load header
1180  '              JR        NC,OOPS     ; carry clear=error
1190  '              LD        HL,HEADER   ; set DE and HL to poin
      t
1200  '              LD        DE,NAME     ; filename/desired name

1210  ' LOOP         LD        A,(DE)      ; check each character
1220  '              CP        (HL)
1230  '              JR        NZ,LOAD     ; if not same, load
1235  '                                    ; next header
1240  '              INC       HL
1250  '              INC       DE
1260  '              LD        A,(DE)
1270  '              CP        0           ; zero indicates the
1275  '                                    ; end of name
1280  '              JR        NZ,LOOP     ; if not end, next char
      '
1290  '              LD        IX,LOADING
1300  '              CALL      SPRINT
1310  '              LD        IX,NAME     ; pick up load address
1320  '              LD        L,(IX+16)
1330  '              LD        H,(IX+17)
1340  '              LD        IX,HEADER   ; pick len from head
```

```
1350  '           LD      E,(IX+24)
1360  '           LD      D,(IX+25)
1370  '           LD      A,&16          ; sync byte
1380  '           CALL    &BCA1
1390  '           JR      NC,OOPS
1400  '           RET
1410  ' SPRINT    PUSH    IX
1420  '           CALL    &BB54
1430  '           LD      A,13
1440  '           CALL    &BB5A
1450  '           LD      A,10
1460  '           CALL    &BB5A
1470  '           POP     IX
1480  ' LOOPS     LD      A,(IX)
1490  '           CP      0
1500  '           RET     Z
1510  '           PUSH    IX
1520  '           CALL    &BB5D
1530  '           POP     IX
1540  '           INC     IX
1550  '           JR      LOOPS
1560  ' OOPS      LD      IX,ERROR
1570  '           CALL    SPRINT
1580  '           LD      A,7
1590  '           CALL    &BB5A
1600  '           RET
1610  '
1620  ' ERROR     TEXT    "I▴beg▴to▴inform▴you▴of▴error",0
1630  ' OK        TEXT    "Continue▴Search?",0
1640  ' SEARCH    TEXT    "Searching.....",0
1650  ' LOADING   TEXT    "Loading.....",0
1660  ' NAME      TEXT    "TEST",0,0,0,0,0,0,0,0,0,0,0,0
1670  '           WORD    40000          ; load add
1680  ' HEADER    RMEM    64             ; 64 zeros
1690  ' END
1700  RETURN
```

```
DD 21 BE 9D CD 61 9D DD 21 AD 9D CD 61 9D CD 18 BB FE 79
28 03 FE 59 C0 11 40 00 21 EC 9D 3E 2C CD A1 BC 30 56 21
EC 9D 11 DA 9D 1A BE 20 D8 23 13 1A FE 00 20 F5 DD 21 CD
9D CD 61 9D DD 21 DA 9D DD 6E 10 DD 66 11 DD 21 EC 9D DD
5E 18 DD 56 19 3E 16 CD A1 BC 30 23 C9 DD E5 CD 54 BB 3E
0D CD 5A BB 3E 0A CD 5A BB DD E1 DD 7E 00 FE 00 C8 DD E5
CD 5D BB DD E1 DD 23 18 EF DD 21 90 9D CD 61 9D 3E 07 CD
5A BB C9 49 20 62 65 67 20 74 6F 20 69 6E 66 6F 72 6D 20
79 6F 75 20 6F 66 20 65 72 72 6F 72 00 43 6F 6E 74 69 6E
75 65 20 53 65 61 72 63 68 3F 00 53 65 61 72 63 68 69 6E
67 2E 2E 2E 2E 2E 00 4C 6F 61 64 69 6E 67 2E 2E 2E 2E 2E
00 54 45 53 54 00 00 00 00 00 00 00 00 00 00 00 00 40 9C
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 20 31
```

Notes As before, NAME holds the name of the file that you want to read in and it's load address. HEADER is a block of 64 bytes for the temporary storage of headers read in from tape. You will note (I hope) that we've use a variation on the SPRINT program to print out messages.

On calling the routine, the messages:

"Searching"
"Continue Search?"

will be displayed. Answering anything but "Y" or "y" at this point will terminate the routine. This question will be asked each time a header is read that is not the header of the file of interest. Reading errors are also flagged with suitable message.

The only other tape handling routine that we are liable to need is a verify function. CVERIFY provides this.

CVERIFY

This routine checks a file of bytes on tape against an area of memory in the computer. It will thus allow you to check that an area of memory has been correctly saved before clearing that area of memory.

Entry Requirements: See Notes.

Exit Conditions: All Corrupt.

Length: 221 Bytes, excluding NAME and HEADER.

```
VERIFY


1000  MEMORY 39999
1010  GOSUB 1030
1020  END
1030   ASSEMBLE
1040  '           ORG       40200
1050  '           LD        IX,SEARCH
1060  '           CALL      SPRINT
1070  ' LOAD      LD        IX,OK
1080  '           CALL      SPRINT
1090  '           CALL      &BB18
1100  '           CP        121
1110  '           JR        Z,YES
1120  '           CP        89
1130  '           RET       NZ
1140  ' YES       LD        DE,64       ;No of bytes in head
1150  '           LD        HL,HEADER   ; put them here
1160  '           LD        A,&2C       ; correct sync byte
1170  '           CALL      &BCA1       ; load header
1180  '           JR        NC,OOPS     ; carry clear=error
1190  '           LD        HL,HEADER   ; set DE/HL to point
1200  '           LD        DE,NAME     ; fname/desired name
1210  ' LOOP      LD        A,(DE)      ; check each character
1220  '           CP        (HL)
1230  '           JR        NZ,LOAD     ; if not same, next hea
      d
1240  '           INC       HL
1250  '           INC       DE
1260  '           LD        A,(DE)
1270  '           CP        0           ; 0 means end name
1280  '           JR        NZ,LOOP     ; if not end, next char
```

```
1290   '          LD       IX,LOADING
1300   '          CALL     SPRINT
1310   '          LD       IX,NAME      ; pick up load address
1320   '          LD       L,(IX+16)
1330   '          LD       H,(IX+17)
1340   '          LD       IX,HEADER    ; pick len from head
1350   '          LD       E,(IX+24)
1360   '          LD       D,(IX+25)
1370   '          LD       A,&16        ; sync byte
1380   '          CALL     &BCA4
1390   '          JR       NC,OOPS
1400   '          RET
1410   ' SPRINT   PUSH     IX
1420   '          CALL     &BB54
1430   '          LD       A,13
1440   '          CALL     &BB5A
1450   '          LD       A,10
1460   '          CALL     &BB5A
1470   '          POP      IX
1480   ' LOOPS    LD       A,(IX)
1490   '          CP       0
1500   '          RET      Z
1510   '          PUSH     IX
1520   '          CALL     &BB5D
1530   '          POP      IX
1540   '          INC      IX
1550   '          JR       LOOPS
1560   ' OOPS     LD       IX,ERROR
1570   '          CALL     SPRINT
1580   '          LD       A,7
1590   '          CALL     &BB5A
1600   '          RET
1610   '
1620   ' ERROR    TEXT     "Verify error!!",0
1630   ' OK       TEXT     "Continue Search?",0
1640   ' SEARCH   TEXT     "Searching.....",0
1650   ' LOADING  TEXT     "Verifying....",0
1660   ' NAME     TEXT     "TEST",0,0,0,0,0,0,0,0,0,0,0,0
1670   '          WORD     40000        ; load add
1680   ' HEADER   RMEM     64           ; 64 0's
1690   ' END
1700   RETURN
```

```
DD 21 B0 9D CD 61 9D DD 21 9F 9D CD 61 9D CD 18 BB FE 79
28 03 FE 59 C0 11 40 00 21 DF 9D 3E 2C CD A1 BC 30 56 21
DF 9D 11 CD 9D 1A BE 20 D8 23 13 1A FE 00 20 F5 DD 21 BF
9D CD 61 9D DD 21 CD 9D DD 6E 10 DD 66 11 DD 21 DF 9D DD
5E 18 DD 56 19 3E 16 CD A4 BC 30 23 C9 DD E5 CD 54 BB 3E
0D CD 5A BB 3E 0A CD 5A BB DD E1 DD 7E 00 FE 00 C8 DD E5
CD 5D BB DD E1 DD 23 18 EF DD 21 90 9D CD 61 9D 3E 07 CD
5A BB C9 56 65 72 69 66 79 20 65 72 72 6F 72 21 21 00 43
6F 6E 74 69 6E 75 65 20 53 65 61 72 63 68 3F 00 53 65 61
72 63 68 69 6E 67 2E 2E 2E 2E 2E 00 56 65 72 69 66 79 69
6E 67 2E 2E 2E 2E 00 54 45 53 54 00 00 00 00 00 00 00 00
00 00 00 00 40 9C 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00
```

Notes 'NAME' should be set up as for the CREAD routine, with the difference that bytes 16 and 17 of NAME should now hold not the load address but the address from which the bytes were saved. Should a difference between the area of memory and the tape file be detected, then an immediate exit of the routine is made, with an error message to signify the fact. The routine only verifies the data block.

That completes this Chapter of Tape Handling Routines. For further information on the Firmware routines, I direct you to the Amstrad "Firmware Technical Manual", which is very useful indeed.

9. BASIC and Machine Code

In this Chapter, we'll see a couple of routines that are designed to help the BASIC programmer, and some detailed notes on the Resident System Extension — the way in which we can add commands to Amstrad BASIC. However, we'll start with a couple of routines that don't really belong in any other Chapter of the book.

TIMESET

The System Variable, TIME, will, when evaluated, return the amount of time that has passed since the computer was first turned on. This does not include such periods of time when interrupts were disabled from within machine code routines, such as periods of tape operations. The TIME variable is a 4 byte value, which is incremented once every 300th of a second. It is thus very useful for timing applications. One feature, though, that I miss is a means by which the TIME variable can be set to any particular time. This sort of command is very common on other machines, such as the BBC Microcomputer (Dare I say such words in these pages?!) and is very useful when the timer is being used for timing short events in a running program. Normally, we have to execute a line such as:

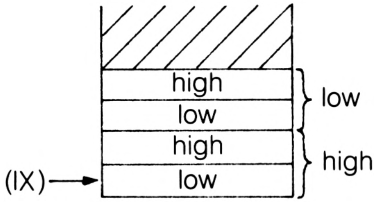
```
T=TIME:GOSUB 1000:PRINT TIME-T
```

whereas if we could set TIME to zero, we could just say:

```
CALL zerotime:GOSUB 1000:PRINT TIME
```

This makes what we're trying to do rather more obvious. The next routine, TIMESET, does this.

Entry Requirements: From BASIC, CALL add,low,high where low is the lower 16 bits of the full 32 bit value and 'high' is the high 16 bits of the full 32 bit value. From a machine code routine, IX points to a parameter block and A holds the value 2.



Parameter Block For TIMESET

Exit Conditions: All registers corrupt. The routine will be exited if the wrong number of parameters are passed in to the routine.

Length: 98 Bytes.

```
TIMESET


1000  MEMORY 39999
1010  GOSUB 1030
1020  END
1030   ASSEMBLE
1040  '             ORG       40200
1050  '             CP        0
1060  '             JR        Z,ZERO      ;if no parameters, zero
1070  '             CP        2
1080  '             JR        NZ,OOPS     ; if not 2 param, error
1090  '             LD        E,(IX+0)
1100  '             LD        D,(IX+1)    ; high part of TIME
1110  '             LD        L,(IX+2)
1120  '             LD        H,(IX+3)    ; low part of TIME
1130  '             CALL      &BD10       ; set TIME
1140  '             RET
1150  ' ZERO        LD        HL,0
1160  '             LD        DE,0
1170  '             CALL      &BD10       ; set TIME to zero
1180  '             RET
1410  ' SPRINT      PUSH      IX
1420  '             CALL      &BB54
1430  '             LD        A,13
1440  '             CALL      &BB5A
1450  '             LD        A,10
1460  '             CALL      &BB5A
1470  '             POP       IX
1480  ' LOOPS       LD        A,(IX)
1490  '             CP        0
1500  '             RET       Z
```

```
1510  '             PUSH      IX
1520  '             CALL      &BB5D
1530  '             POP       IX
1540  '             INC       IX
1550  '             JR        LOOPS
1560  ' OOPS        LD        IX,ERROR
1570  '             CALL      SPRINT
1580  '             LD        A,7
1590  '             CALL      &BB5A
1600  '             RET
1610  '
1620  ' ERROR       TEXT      "Parameter Error!",0
1690  ' END
1700  RETURN
```

```
FE 00 28 14 FE 02 20 3C DD 5E 00 DD 56 01 DD 6E 02 DD 66
03 CD 10 BD C9 21 00 00 11 00 00 CD 10 BD C9 DD E5 CD 54
BB 3E 0D CD 5A BB 3E 0A CD 5A BB DD E1 DD 7E 00 FE 00 C8
DD E5 CD 5D BB DD E1 DD 23 18 EF DD 21 59 9D CD 2A 9D 3E
07 CD 5A BB C9 50 61 72 61 6D 65 74 65 72 20 45 72 72 6F
72 21 00
```

Notes The fact that TIME is a 32 bit variable makes passing such a value via a standard CALL statement rather difficult. This is why the new TIME value is passed to the machine code routine in two parts. Thus to set TIME to 1ØØ, we would execute a command such as:

CALL address, 1ØØ,Ø

with 'low' being first.

The bytes given above are for address 41ØØØ, but the routine can be relocated with little trouble. Because I often want to set the TIME variable to zero, I decided to make it a special case. CALL address on its own will set the TIME variable to zero. A few notes will be given here about values to pass as parameters. The value put in to TIME for any particular combination of 'high' and 'low' will be:

TIME = low+(65536*high)

Thus, the command:

CALL address,Ø,2

will set TIME to (2*65536)+Ø, which is 137Ø2. Of course, setting the TIME variable to a very high value will lead to it eventually clocking through zero and starting again.

Cleaning up

Badly written programs of any description are a bit like me; a lot of cleaning up is needed after they've been executed, assuming that you haven't crashed the system. For example, a common problem is accidentally generating a totally unreadable combination of INK and PAPER, or a sound that goes on, and on, and on . . .

Well, there are a few routines in the firmware that can be called to clean things up a little after such events. These are as follows:

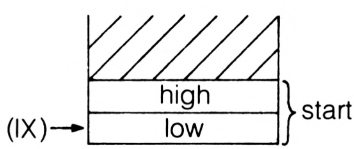
&BB4E	Text Screen Handling
&BBØØ	Keyboard
&BBBA	Graphics Screen Handling
&BCØ2	Screen Pack
&BC65	Cassette Handling
&BCA7	Sound Handling
&BD37	Jump Block Restore

Of these, &BCØ2, &BCA7 and &BD37 are likely to be the most useful. &BCØ2 is extremely useful, in that it sets the colours back to their original states. Very useful after you've managed to get the pallette of the computer into a bit of a mess by fiddling around! &BD37 is only likely to be useful if you're doing some advanced work that involves you in altering Jump Block Entries. This routine sets them back to their original conditions. Calling &BCA7 will provide relief from a sound that doesn't want to stop. It can also be used in running programs to 'flush' the sound buffer and so stop any sounds being played at that time.

ROMREAD

As was mentioned earlier in this book, RAM overlays both 16k blocks of ROM. POKE and PEEK both operate on RAM locations in these areas only. However, it's occasionally interesting to have a look in ROM to see what error messages you haven't discovered yet, command table structures and even to see how the professional programmers have done a particular piece of code. This routine lists the contents of ROM locations in blocks of 20 bytes. It was originally written for use in screen mode 2. Firmware routines are used to allow us to access the upper and lower ROMs.

Entry Requirements: From BASIC, CALL address,start where start is the address of the first location whose contents are to be listed. From machine code, IX point to a two byte parameter block and A =1.



ROMREAD Parameter Block

Exit Conditions: All registers corrupt.

Length: 161 Bytes.

ROMREAD

```
1000  MEMORY 39999
1010  GOSUB 1030
1020  END
1030  |ASSEMBLE
1040  '            ORG     40200
1050  '            LD      B,20         ; print 20 bytes
1060  '            LD      L,(IX+0)
1070  '            LD      H,(IX+1)     ; pick up start address

1080  ' OLOOP      PUSH    BC
1090  '            PUSH    HL           ; preserve the reg's
1100  '            CALL    &B906        ; enable lower ROM
1110  '            PUSH    AF           ; save ROM status
1120  '            CALL    &B900        ; enable upper ROM
1130  '            LD      A,(HL)
1140  '            LD      (STORE),A    ; get byte and store it

1150  '            POP     AF           ; get ROM status
1160  '            CALL    &B90C        ; put ROMs back to norm

1170  '            POP     HL
1180  '            PUSH    HL
1190  '            CALL    PNUMHL       ; print address in hex
1200  '            LD      IX,SPACE     ; print spaces
1210  '            CALL    SPRINT
1220  '            LD      A,(STORE)    ; get byte read
1230  '            CALL    PNUMA        ; print byte in hex
1231  '            LD      IX,SPACE
1232  '            CALL    SPRINT       ; print spaces
1233  '            LD      A,(STORE)    ; now print as a
1234  '            CALL    &BB5D        ; char, inc cntrl codes

1240  '            LD      A,13
1250  '            CALL    &BB5A
1260  '            LD      A,10
1270  '            CALL    &BB5A        ; carriage return
1280  '            POP     HL
1290  '            INC     HL           ; next byte
1300  '            POP     BC
1310  '            DJNZ    OLOOP        ; 20 bytes yet?
1320  ' SPRINT     PUSH    IX
1330  '            CALL    &BB54
1340  '            POP     IX
1350  ' LOOPS      LD      A,(IX)
1360  '            CP      0
1370  '            RET     Z
1380  '            PUSH    IX
1390  '            CALL    &BB5D
1400  '            POP     IX
1410  '            INC     IX
1420  '            JR      LOOPS
1430  ' PNUMA      LD      B,0           ; these routines have
1440  '            LD      C,A           ; already documented
1450  '            RR      A
1460  '            RR      A
```

```
1470  '            RR        A
1480  '            RR        A
1490  ' PRINLO     AND       &0F
1500  '            CP        &0A
1510  '            JR        NC,ATOF
1520  '            ADD       A,&30
1530  '            PUSH      BC
1540  '            CALL      &BB5A
1550  '            JR        OUT
1560  ' ATOF       ADD       A,&37
1570  '            PUSH      BC
1580  '            CALL      &BB5A
1590  ' .OUT       POP       BC
1600  '            LD        A,B
1610  '            CP        1
1620  '            RET       Z
1630  '            LD        A,C
1640  '            LD        B,1
1650  '            JR        PRINLO
1660  ' PNUMHL     LD        A,H
1670  '            CALL      PNUMA
1680  '            LD        A,L
1690  '            CALL      PNUMA
1700  '            RET
1710  '
1720  ' SPACE      TEXT      "▲▲▲▲▲▲▲▲▲▲▲▲▲▲",0
1725  ' STORE      BYTE      0
1730  ' END
1740  RETURN
```

```
06 14 DD 6E 00 DD 66 01 C5 E5 CD 06 B9 F5 CD 00 B9 7E 32
A8 9D F1 CD 0C B9 E1 E5 CD 90 9D DD 21 99 9D CD 4F 9D 3A
A8 9D CD 67 9D DD 21 99 9D CD 4F 9D 3A A8 9D CD 5D BB 3E
0D CD 5A BB 3E 0A CD 5A BB E1 23 C1 10 C1 DD E5 CD 54 BB
DD E1 DD 7E 00 FE 00 C8 DD E5 CD 5D BB DD E1 DD 23 18 EF
06 00 4F CB 1F CB 1F CB 1F CB 1F E6 0F FE 0A 30 08 C6 30
C5 CD 5A BB 18 06 C6 37 C5 CD 5A BB C1 78 FE 01 C8 79 06
01 18 E2 7C CD 67 9D 7D CD 67 9D C9 20 20 20 20 20 20 20
20 20 20 20 20 20 20 00 00
```

Notes 3 ROM routines are important here, and we've looked at two of them before, when we examined the VARCHAR routine in Chapter 3. The Routine at &B9ØØ pages in the Upper ROM that contains the BASIC Interpreter, and returns the ROM status in the A register.

These three routines are not, strictly speaking, firmware routines because they are in RAM. A second's thought will explain why this is so. There is little point in having routines to page in and out ROM in the actual ROMs; if one is needed and the relevant ROM is paged out then we would have problems!

The routine, when called with a start address, will print out the character whose ASCII code is in the addresses being examined, even if the ASCII code read from the address is that of a control code. We can do this by using the routine at &BB5D instead of &BB5A. Because

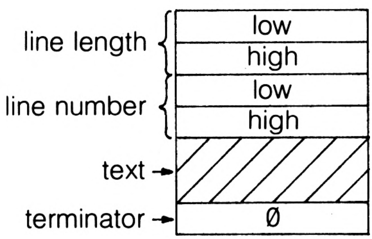
ROMREAD makes extensive use of subroutines, it is very difficult to relocate without the aid of an Assembler package. The bytes given above are for address 4Ø2ØØ. If you do decide to relocate the program, then it is important to keep it all within the 'Memory Pool' area of RAM. Otherwise, when the ROMs were paged in to gain access to their contents, part or all of your program would be paged out! Not a desirable state of affairs.

The routine can, of course, be used to read the contents of RAM that are not overlaid by ROM.

The next routine is called FIND, and is used to scan through a BASIC program to find text or variable names. It then prints out the line numbers at which a particular 'target' string has been found in programs. Before we look at the routine, a short examination of how the text of a BASIC program is stored in memory might be useful.

BASIC LINE Structure

A typical line of BASIC is in the form:



The line length is the number of bytes in the complete line, including the line length bytes, line number and terminator bytes. The byte 'Ø' terminator acts to separate one line from another. The end of a program can be located by searching through the program text for a line length entry that is set to zero and a line number entry that is set to zero.

Altering line lengths and line numbers by POKEing in to the appropriate positions in memory can be rather interesting; however, you can also appear to lose a program with the line number POKEing. The program is still there, but you have to provide the correct line length. One interesting point is that you can often render a program unreadable, and still run it. However, we are wandering off the point a little here. The BASIC program text starts at address 368 with the low byte of the line length of the first program line.

One point here is that, unlike on some machines, first line REM statements are not the best place to save machine code, mainly due to the fact that it would be in an area of RAM overlaid by ROM. This would give rise to problems if ROM were to be paged in.

Within the actual text of the line, BASIC keywords and functions are stored as single bytes with values between 128 and 255 which are called TOKENS. For example, ABS has the value 255, AFTER is 128 and REM is 197. Text after a REM token, or within the ' " ' of a string assignment or a PRINT statement, is stored as a sequence of ASCII codes. Any assignments in a program line, such as:

```
100 a$="fred"
```

are quite interesting; if the address of a$ is passed over to a machine code routine, using CALL address,@a$, then the value given as the address will be in the body of the program, in line 100. The interesting thing is that if we modify the string within our machine code routine, then line 100 will be modified. We saw this in action in the INSTRING program. The address of any variable can be obtained by simply typing in:

```
PRINT @variable-name
```

You do not have to include it as part of a CALL statement; @ can be used separately. However, information such as the address of variables is probably of minimal use to us in practical programming, due to the fact that CALL sorts all the details out for us.

What is more use to us with regard to FIND is the way in which the actual variable names are stored in the program lines. Variable names are stored in ASCII, with their last character having 128 added to its ASCII code. Thus for single letter variable names, the letter has 128 added to its ASCII code. By last character, I mean the last letter of the variable name, NOT the Type Identifier if one is present. Thus the variable name 'fred' would be stored as:

```
f   102
r   114
e   101
d   228  (128+100)
```

For our purposes, this is really all we need to know about the storage of variable names. You can probably see one big problem with any search for variable names in Amstrad BASIC programs; the last character, with 128 added to its ASCII code, now has a code in the same range as the BASIC Tokens. This can, and, with FIND occasionally does, lead to problems, particularly when the variable name for which we are searching has only got 1 letter in it. The routine will occasionally find, when searching for single letter variable names, lines which contain a particular BASIC Token instead of the variable. However, with longer names there is no problem.

FIND

A routine to find text or variable names within the body of a BASIC program. Line numbers referring to the location of the 'target' text or variable name are printed out.

Entry Requirements: From BASIC: CALL address,@a$,mode where 'mode' defines whether the search will be made for a variable name (with suitable modified last character) when mode=Ø, or normal text when mode=1. When mode=1 the last character of the target string in a$ is not modified and so variable names will not be found. See Notes for further details.

Exit Conditions: Not Applicable.

Length: 334 Bytes.

```
FIND


1000  MEMORY 39999
1010  GOSUB 1050
1020  F=23
1030  SLINE=34
1040  END
1050  |ASSEMBLE
1060  '              ORG       42000
1070  '              CP        2
1080  '              JR        NZ,PARAMS
1090  '              LD        L,(IX+2)
1100  '              LD        H,(IX+3)
1110  '              LD        A,(HL)
1120  '              LD        (LENGTH),A     ; length/descriptor
1130  '              INC       HL
1140  '              LD        C,(HL)
1150  '              INC       HL
1160  '              LD        B,(HL)
1170  '              LD        (STRING),BC    ; string address
1180  '              LD        A,(IX+0)
1190  '              CP        1
1200  '              JR        Z,OK           ; if straight text go

1210  '              LD        A,(LENGTH)     ; otherwise modify
1220  '              LD        E,A            ; the last character
1230  '              LD        D,0            ; of the var.name
1240  '              DEC       DE             ; by adding 128 to
1250  '              LD        HL,(STRING)    ; it and then replace

1260  '              ADD       HL,DE
1270  '              LD        A,(HL)
1280  '              ADD       A,128
1290  '              LD        (HL),A         ; it in memory
1300  '              LD        A,(LENGTH)
1310  '              CP        1              ; if var.name only 1
1320  '              CALL      Z,POSSPROB     ; char print message
1330  ' OK           LD        IX,368         ; start of prog in
1335  '                                       ; Tape system
```

```
1340  ' LOOP     LD      E,(IX+0)
1350  '          LD      D,(IX+1)      ; DE = line length
1360  '          LD      L,(IX+2)      ; HL = line number
1370  '          LD      H,(IX+3)
1380  '          LD      (LINE),HL
1390  '          LD      A,L
1400  '          OR      H
1410  '          JR      Z,FINISH      ; if zero we've done
1420  '          CALL    &BB1B
1430  '          JR      C,FINISH      ; if key pressed
1435  '                                ; we've done
1440  '          CALL    SLINE         ; scan the line
1450  '          PUSH    IX
1460  '          POP     HL
1470  '          DEC     DE
1480  '          ADD     HL,DE         ; update HL to point
1490  '          INC     HL            ; start of next line
1500  '          PUSH    HL            ; transfer to IX
1510  '          POP     IX
1520  '          JR      LOOP          ; round again
1530  ' PARAMS   JR      PARAM2        ; needed for relative

1535  '                                ; jump!
1540  ' SLINE    PUSH    IX
1550  '          POP     HL            ; start of line in HL

1560  '          PUSH    IX
1570  '          PUSH    DE
1580  '          POP     BC            ; length in BC
1590  '          DEC     BC
1600  '          DEC     BC
1610  '          DEC     BC            ; reduce line by 4
1620  '          DEC     BC            ; so that only text
1625  '                                ; is scanned
1630  '          PUSH    DE
1640  '          LD      DE,04
1650  '          ADD     HL,DE         ; point HL to start
1655  '                                ; of text
1660  ' LOOP1    LD      DE,(STRING)   ; address of target
1665  '                                ; in DE
1670  '          PUSH    BC            ; preserve line leng
1680  '          LD      A,(DE)
1690  '          CP      (HL)
1700  '          JR      Z,YES         ; if two char. match,

1710  ' NOTOK    POP     BC            ; decrement counter,
1720  '          DEC     BC
1730  '          INC     HL            ; point to next char.

1740  '          LD      A,B
1750  '          OR      C
1760  '          JR      NZ,LOOP1      ; if line not done
1765  '                                ; carry on
1770  '          POP     DE
1780  '          POP     IX            ; restore registers
1790  '          RET
1800  ' YES      LD      A,(LENGTH)
1810  '          LD      B,A
1820  ' LOOP2    LD      A,(DE)        ; scan rest of target

1830  '          CP      (HL)          ; to see if match
1840  '          JR      NZ,NOTOK      ; if not go back
1850  '          INC     HL
```

```
1860 '          INC     DE
1870 '          DJNZ    LOOP2
1880 '          CALL    FOUND          ; get here it's a hit

1890 '          JR      NOTOK
1900 ' FOUND    LD      A,10
1910 '          CALL    &BB5A
1920 '          LD      A,13
1930 '          CALL    &BB5A          ; CR + LF
1940 '          PUSH    IX
1950 '          PUSH    HL
1960 '          PUSH    DE
1970 '          PUSH    BC
1980 '          LD      HL,(LINE)
1990 '          CALL    PDECHL
2000 '          POP     BC
2010 '          POP     DE
2020 '          POP     HL
2030 '          POP     IX
2040 '          RET
2050 ' PARAM2   LD      IX,PARA
2060 '          CALL    SPRINT
2070 '          RET
2080 ' POSSPROB LD      IX,MESS1
2090 '          CALL    SPRINT
2100 '          RET
2110 ' FINISH   RET
2120 '
2130 ' SPRINT   PUSH    IX           ; routines documented
2140 '          CALL    &BB54        ; elsewhere
2150 '          LD      A,10
2160 '          CALL    &BB5A
2170 '          LD      A,13
2180 '          CALL    &BB5A
2190 '          LD      A,7
2200 '          CALL    &BB5A
2210 '          POP     IX
2220 ' LOOPS    LD      A,(IX)
2230 '          CP      0
2240 '          RET     Z
2250 '          PUSH    IX
2260 '          CALL    &BB5D
2270 '          POP     IX
2280 '          INC     IX
2290 '          JR      LOOPS
2300 ' PDECHL   LD      DE,10000
2310 '          CALL    PDECH
2320 '          LD      DE,1000
2330 '          CALL    PDECH
2340 '          LD      DE,100
2350 '          CALL    PDECH
2360 '          LD      DE,10
2370 '          CALL    PDECH
2380 '          LD      DE,1
2390 ' PDECH    XOR     A
2400 ' LOOP4    SCF
2410 '          CCF
2420 '          SBC     HL,DE
2430 '          JR      C,PDOUT
2440 '          INC     A
2450 '          JR      LOOP4
2460 ' PDOUT    ADD     HL,DE
2470 '          ADD     A,&30
```

```
2480  '            PUSH      HL
2490  '            CALL      &BB5A
2500  '            POP       HL
2510  '            RET
2520  '
2530  ' MESS1       TEXT      "May give some odd results!!",0
2540  ' PARA        TEXT      "Parameter Error!!",0
2550  ' LINE        WORD      0000
2560  ' STRING      WORD      0000
2570  ' LENGTH      BYTE      00
2580  ' END
2590  RETURN
```

```
FE 02 20 5A DD 6E 02 DD 66 03 7E 32 58 A5 23 4E 23 46 ED
43 56 A5 DD 7E 00 FE 01 28 17 3A 58 A5 5F 16 00 1B 2A 56
A5 19 7E C6 80 77 3A 58 A5 FE 01 CC C8 A4 DD 21 70 01 DD
5E 00 DD 56 01 DD 6E 02 DD 66 03 22 54 A5 7D B4 28 75 CD
1B BB 38 70 CD 70 A4 DD E5 E1 1B 19 23 E5 DD E1 18 DA 18
50 DD E5 E1 DD E5 D5 C1 0B 0B 0B 0B D5 11 04 00 19 ED 5B
56 A5 C5 1A BE 28 0B C1 0B 23 78 B1 20 F0 D1 DD E1 C9 3A
58 A5 47 1A BE 20 ED 23 13 10 F8 CD A5 A4 18 E4 3E 0A CD
5A BB 3E 0D CD 5A BB DD E5 E5 D5 C5 2A 54 A5 CD F8 A4 C1
D1 E1 DD E1 C9 DD 21 42 A5 CD D1 A4 C9 DD 21 26 A5 CD D1
A4 C9 C9 DD E5 CD 54 BB 3E 0A CD 5A BB 3E 0D CD 5A BB 3E
07 CD 5A BB DD E1 DD 7E 00 FE 00 C8 DD E5 CD 5D BB DD E1
DD 23 18 EF 11 10 27 CD 13 A5 11 E8 03 CD 13 A5 11 64 00
CD 13 A5 11 0A 00 CD 13 A5 11 01 00 AF 37 3F ED 52 38 03
3C 18 F7 19 C6 30 E5 CD 5A BB E1 C9 4D 61 79 20 67 69 76
65 20 73 6F 6D 65 20 6F 64 64 20 72 65 73 75 6C 74 73 21
21 00 50 61 72 61 6D 65 74 65 72 20 45 72 72 6F 72 21 21
00 00 00 00 00 00
```

Notes Due to its extensive use of subroutines, this program can only be relocated with difficulty. All references to subroutine addresses will need to be altered. Use has been made of routines like PDECHL and SPRINT which we saw earlier on in the book. The bytes in the above listing are for address 42000. The program is relatively simple to use.

a$="fred":CALL 42000,@a$,0

will search for a variable name 'fred'. Don't put in the type identifier if looking for something like 'fred%' or 'fred$'. Simply leave it out. FIND will then come up with all occurrences of a variable name 'fred', irrespective of the type. If you attempt to search for a single letter variable name, you will be warned by the program that this can occasionally give rise to some odd results. During a search, if you want to finish, simply press any key. This will terminate the search.

a$="fred":CALL 42000,@a$,1

will search for a piece of text with 'fred' in it. This could be a PRINT or REM statement, or could be part of a variable name, such as 'freda'. 'fred' is in this, and will be detected by FIND. Using the routine with mode=0 will return to BASIC with the last character of a$ being corrupted by having 128 added to it's ASCII code. This does not happen when mode=1.

We saw earlier in this Chapter how the end of a BASIC program is indicated by the presence of line length and line number set to zero. The next routine, PLENGTH, uses this fact to give the length, in bytes, of a BASIC program.

PLENGTH

Prints the length of a BASIC program.

Entry Requirements: CALL address

Exit Conditions: Not Applicable.

Length: 93 Bytes.

```
PLENGTH


1000  MEMORY 39999
1010  GOSUB 1050
1020  F=23
1030  SPLINE=34
1040  END
1050  |ASSEMBLE
1060  '           ORG      42000
1330  ' OK        LD       IX,368       ; start prog/Tape
1335  '           LD       BC,0
1340  ' LOOP      LD       E,(IX+0)
1350  '           LD       D,(IX+1)     ; DE = line length
1360  '           LD       L,(IX+2)     ; HL = line number
1370  '           LD       H,(IX+3)
1390  '           LD       A,L
1400  '           OR       H
1410  '           JR       Z,FINISH     ; if zero we've done
1420  '           PUSH     HL
1421  '           PUSH     BC
1422  '           POP      HL           ; BC into HL
1423  '           ADD      HL,DE        ; add line len to HL
1424  '           PUSH     HL
1425  '           POP      BC           ; get BC/updated
1426  '           POP      HL
1450  '           PUSH     IX
1460  '           POP      HL
1470  '           DEC      DE
1480  '           ADD      HL,DE        ; update HL to point
1490  '           INC      HL           ; start of next line
1500  '           PUSH     HL           ; transfer to IX
1510  '           POP      IX
1520  '           JR       LOOP         ; round again
2110  ' FINISH    PUSH     BC
2120  '           POP      HL           ; BC in to HL
2130  '           CALL     PDECHL       ; print it
2140  '           RET
2300  ' PDECHL   LD       DE,10000
2310  '          CALL     PDECH
2320  '          LD       DE,1000
2330  '          CALL     PDECH
```

```
234Ø  '           LD        DE,1ØØ
235Ø  '           CALL      PDECH
236Ø  '           LD        DE,1Ø
237Ø  '           CALL      PDECH
238Ø  '           LD        DE,1
239Ø  ' PDECH     XOR       A
24ØØ  ' LOOP4     SCF
241Ø  '           CCF
242Ø  '           SBC       HL,DE
243Ø  '           JR        C,PDOUT
244Ø  '           INC       A
245Ø  '           JR        LOOP4
246Ø  ' PDOUT     ADD       HL,DE
247Ø  '           ADD       A,&3Ø
248Ø  '           PUSH      HL
249Ø  '           CALL      &BB5A
25ØØ  '           POP       HL
251Ø  '           RET
258Ø  ' END
259Ø  RETURN
```

```
DD 21 7Ø Ø1 Ø1 ØØ ØØ DD 5E ØØ DD 56 Ø1 DD 6E Ø2 DD 66 Ø3
7D B4 28 12 E5 C5 E1 19 E5 C1 E1 DD E5 E1 1B 19 23 E5 DD
E1 18 DE C5 E1 CD 3F A4 C9 11 1Ø 27 CD 5A A4 11 E8 Ø3 CD
5A A4 11 64 ØØ CD 5A A4 11 ØA ØØ CD 5A A4 11 Ø1 ØØ AF 37
3F ED 52 38 Ø3 3C 18 F7 19 C6 3Ø E5 CD 5A BB E1 C9
```

Notes The routine is again a little difficult to relocate due to the use of subroutines. The above bytes are for address 42ØØØ. To use the routine, simply CALL it when required. The program length will be printed to the screen, and should there be no program in the computer then a value of Ø will be printed.

Resident System Extensions

The assembler that was used to produce the listings in this book was on a ROM chip, and was invoked by typing in the command

|ASSEMBLE

The vertical line, accessed from the keyboard by SHIFT @ informs the BASIC Interpreter that the text following it is to be treated as an extra command, and that details on how the command is to be processed will be found in RAM or ROM. The facility that allows us to add commands like this to the BASIC command structure is called The Resident System Extension, or RSX for short. Essentially, it is a way of calling machine code routines by name, rather than having to remember the address. This is clearly useful if you've got several different routines that you want to call in your program from BASIC. Parameters can be passed to RSX routines in a way that is virtually identical to the way in which parameters can be passed to the machine code routines accessed by CALL. RSX commands can pass values back to BASIC variables using the '@' function; a line such as:

|GET,@character%

could, for example, wait for a key press and return the ASCII code of the key pressed in the variable 'character%'. A line such as

character% = |GET

isn't possible. So, RSX commands are not true extensions to BASIC, they're more like named CALL routines. However, this should not detract from their usefulness.

The BASIC Interpreter must be informed of the presence of the RSX commands, and we're lucky in that all we need do is call one operating system Firmware routine. To show this in action, we'll add a couple of RSX commands, |CLS and |FRAME.

|CLS will clear the screen and restores the usual 'turn on' colour pallette.

|FRAME will cause processing to halt until the next frame of the display has been drawn. This is useful in graphics programs, where it can help cut down flickering.

Neither of these commands accepts parameters, but we'll shortly add a command that does.

At the heart of the RSX System are two tables, the jump table and the name table. I'll just give enough information here to allow you to use the system. If you want full details you should consult the "Technical Firmware Manual" from Amsoft.

The Jump Table

This holds the addresses to which the various RSX commands added pass control. The addresses are stored as part of a Z-80 JP instruction.

```
TABLESTART    WORD    names     ; address of name table
              JP      routine1  ; first routine
              JP      routine2  ; second routine
              JP      routine3  ; and so on
```

The name table will be looked at shortly. One entry in the jump table is needed for each RSX command that you're adding, and the order in which they appear is the same order as that in which commands appear in the RSX name table.

The name table

This holds the actual names of the RSX commands that you want to add. The start address of this table is stored, low byte first, in the first two bytes of the jump table.

```
names   TEXT   name_of_command1   ; name of routine1
        TEXT   name_of_command2   ; name of routine2
        TEXT   name_of_command3   ; name of routine3
        BYTE   Ø                  ; terminates table
```

There is one important point to note about this table. That is that the last character of each command name has 128 added to its ASCII code. A further point to note is the order of the entries; 'name-of-command1' when encountered will cause a jump to 'routine1' and so on. Finally, all entries in the Name table should be in upper case. All RSX commands entered in to the machine are converted to upper case by the BASIC Interpreter, so we might as well put the command names in the table in upper case as well, remembering that the last character has to be modified. The Interpreter also requires 4 bytes of workspace, which can be anywhere in the Memory Pool. Once set up, the tables are 'activated', and the commands added to the command structure, by a call to address &BCD1 with the address of the workspace in HL and the address of the start of the jump table in BC. This call should only be done once. More than once for the same table seems to cause the machine to occasionally lock up.

Other tables of commands can also be added, one after the other if you want. Of course, you should ensure that two or more RSX commands don't have the same name!

RSX1

This routine adds two RSX commands, |CLS and |FRAME.

Entry Requirements: CALL address, where address is the enabling routine address. This should only be called once.

Exit Conditions: Not Applicable.

Length: 44 Bytes, including RSX Tables.

```
RSX1


1000  MEMORY 39999
1010  GOSUB 2000
1020  CALL 40200
1030  END
2000  |ASSEMBLE
2010  '          ORG       40200
2020  '          LD        BC,TABLE
2030  '          LD        HL,WORK
2040  '          CALL      &BCD1          ; set table up
2050  '          RET
2060  ' FRAME    CALL      &BD19          ; routine for frame
2070  '          RET
2080  ' CLS      CALL      &BC02          ; routine for CLS
```

```
2090  '          LD         A,12
2100  '          CALL       &BB5A
2110  '          RET
2120  ' TABLE    WORD       NAMES         ; address of name table
2130  '          JP         FRAME          ; jump to frame
2140  '          JP         CLS            ; jump to cls
2150  ' NAMES    TEXT       "FRAM",197     ; FRAME with last char.

2155  '                                    ; modified
2160  '          TEXT       "CL",211       ; CLS with last char.
2165  '                                    ; modified
2170  '          BYTE       0              ; terminator
2180  ' WORK     RMEM       4              ; workspace
2190  ' END
2200  RETURN
```

```
01 1F 9D 21 30 9D CD D1 BC C9 CD 19 BD C9 CD 02 BC 3E 0C
CD 5A BB C9 27 9D C3 12 9D C3 16 9D 46 52 41 4D C5 43 4C
D3 00 F8 A5 1F 9D
```

Notes Once the tables have been set up by the enable routine, the two commands |FRAME and |CLS will be enabled.

The passing of parameters to the RSX routines is easy. It is essentially identical to passing routines with CALL. On entry to one of the RSX routines, the IX register points to a parameter block and A holds the number of parameters passed with the RSX command. The arrangement of parameters in the parameter block is the same as for the CALL statement. The next routine, |PAUSE,n, demonstrates this.

PAUSE

An RSX command that causes a delay of about n/50 seconds. It does this by repeatedly waiting for the next display frame to be drawn, a process that occurs once every 50th of a second. The delay can also be terminated by pressing a key.

Entry Requirements: |PAUSE,n where n is the desired delay, in 1/50 of a second, between 1 and 65535.

Exit Conditions: Not Applicable.

Length: 105 Bytes.

```
PAUSE


1000  MEMORY 39999
1010  GOSUB 2000
1020  CALL 40200
1030  END
2000   |ASSEMBLE
2010  '          org        40200
2020  '          LD         BC,TABLE
```

```
2Ø3Ø  '           LD     HL,WORK
2Ø4Ø  '           CALL   &BCD1          ; set table up
2Ø5Ø  '           RET
2Ø51  ' .PAUSE    CP     1
2Ø52  '           JR     NZ,ERROR       ; error on > 1 param
2Ø53  '           LD     C,(IX)
2Ø54  '           LD     B,(IX+1)       ; dur of pause in BC
2Ø55  ' LOOP      PUSH   BC
2Ø56  '           CALL   &BD19           ; pause
2Ø57  '           CALL   &BB1B           ; key down?
2Ø58  '           JR     C,FINISH        ; if yes exit
2Ø59  '           POP    BC              ; otherwise.....
2Ø6Ø  '           DEC    BC              ; decrease BC and
2Ø61  '           LD     A,B
2Ø62  '           OR     C
2Ø63  '           JR     NZ,LOOP         ; if not zero, again
2Ø64  '           RET                    ; all done, finish
2Ø65  ' FINISH    POP    BC              ; here if key pressed
2Ø66  '           RET
2Ø67  ' ERROR     LD     IX,MESS1        ; print error message
2Ø68  '           LD     A,7
2Ø69  '           CALL   &BB5A
2Ø7Ø  ' LOOP1     LD     A,(IX)
2Ø71  '           CP     Ø
2Ø72  '           JR     Z,DONE
2Ø73  '           CALL   &BB5A
2Ø74  '           INC    IX
2Ø75  '           JR     LOOP1
2Ø76  ' DONE      LD     A,13
2Ø77  '           CALL   &BB5A
2Ø78  '           LD     A,1Ø
2Ø79  '           CALL   &BB5A
2Ø8Ø  '           RET
212Ø  ' TABLE     WORD   NAMES         ; address of name table
213Ø  '           JP     PAUSE
215Ø  ' NAMES     TEXT   "PAUS",197
217Ø  '           BYTE   Ø              ; terminator
218Ø  ' WORK      RMEM   4              ; workspace
2185  ' MESS1     TEXT   "Parameter Error!!",Ø
219Ø  ' END
22ØØ  RETURN
```

```
Ø1 5Ø 9D 21 5B 9D CD D1 BC C9 FE Ø1 2Ø 18 DD 4E ØØ DD 46
Ø1 C5 CD 19 BD CD 1B BB 38 Ø7 C1 ØB 78 B1 2Ø F1 C9 C1 C9
DD 21 5F 9D 3E Ø7 CD 5A BB DD 7E ØØ FE ØØ 28 Ø7 CD 5A BB
DD 23 18 F2 3E ØD CD 5A BB 3E ØA CD 5A BB C9 55 9D C3 12
9D 5Ø 41 55 53 C5 ØØ F8 A5 5Ø 9D 5Ø 61 72 61 6D 65 74 65
72 2Ø 45 72 72 6F 72 21 21 ØØ
```

Notes An instruction such as

|PAUSE,5Ø

will, once the command has been activated by calling the routine, cause a delay of about 1 second. A keypress during this time will also cause an exit. Try:

P=TIME: PAUSE,5Ø:PRINT TIME-P

which returns the duration of the pause in 1/3ØØths of a second.

Appendix 1.
Control Code Effects

CODE	EFFECT
Ø	No effect.
1	Needs 1 parameter, a value between Ø and 255. The symbol given by the parameter value is printed. This allows the symbols that are associated with characters Ø to 31 to be printed, rather than treated as control codes.
2	Turn off text cursor.
3	Turn on text cursor.
4	One parameter, which is the screen mode. Thus PRINT CHR$(4)+CHR$(1) will set Mode 1.
5	One parameter, between Ø and 255. The parameter is the ASCII code of a character that you want to print to the graphics cursor.
6	Enable the text screen.
7	Bleep.
8	Move cursor back one space.
9	Move cursor forward one space.
1Ø	Move cursor down one line.
11	Move cursor up one line.
12	Clear text window.
13	Move cursor to left of current line.
14	One parameter, which is treated as the Paper Ink number.
15	One parameter, which is treated as the Pen Ink number.
16	Delete the character under the text cursor (same as CLR).
17	Clear from left edge of window to the current character position.
18	Clear from the current character position to the right edge of the window.
19	Clear from the start of the window to the current character position.
2Ø	Clear from the current character position to the end of the window.

Control Codes 16-2Ø all clear the current character position as well as the rest. The characters are cleared to the text paper colour.

21	Turn off the text screen.
22	One parameter. Ø disables transparent mode and 1 enables transparent mode.
23	One parameter, which sets the graphics ink mode. 1 XOR Mode. 2 AND Mode 3 OR Mode.
24	Exchange pen and paper inks.
25	9 parameters. It is the equivalent of the SYMBOL command. The first parameter is the ASCII code of the symbol to be defined, and the next 8 are the definitions for each row of the character. I've found that this code sometimes causes a little trouble.
26	Same as a WINDOW command. Has 4 parameters. The first two parameters specify the left and right hand edges of the window. It doesn't matter which order you put the parameters in, as the smallest is always taken as the left edge. The next two parameters are the top and bottom rows of the window, the smallest value being the top row, the other being the bottom.
27	No Effect.
28	3 parameters. Sets an ink to a pair of colours. The first parameter is the ink no., the second two are the colours.
29	Two parameters. Same as a BORDER command, the two parameters being the colours.
3Ø	Returns cursor to top left of screen window.
31	Two parameters. Same as a LOCATE command. This sets the text cursor to position x,y where x is the first parameter and y is the second parameter.

All these control codes can be passed through the CPRINT routine, or through &BB5A. Note that the firmware routine called at &BB5D does not act on these control codes, but prints the symbol associated with them instead.

Appendix 2.

Instructions and Op-codes

MNEMONIC	HEXADECIMAL
ADC A, (HL)	8E
ADC A, (IX+dis)	DD 8E XX
ADC A,(IY+dis)	FD 8E xx
ADC A,A	8F
ADC A,B	88
ADC A,C	89
ADC A,D	8A
ADC A,n	CE XX
ADC A,E	8B
ADC A,H	8C
ADC A,L	8D
ADC HL,BC	ED 4A
ADC HL,DE	ED 5A
ADC HL,HL	ED 6A
ADC HL,SP	ED 7A
ADD A, (HL)	86
ADD A,(IX+dis)	DD 86XX
ADD A,(IY+dis)	FD 86XX
ADD A,A	87
ADD A,B	80
ADD A,C	81
ADD A,D	82
ADD A,n	C6 XX
ADD A,E	83
ADD A,H	84
ADD A,L	85
ADD HL,BC	09
ADD HL,DE	19
ADD HL,HL	29
ADD HL,SP	39
ADD IX,BC	DD 09
ADD IX,DE	DD 19
ADD IX,IX	DD 29
ADD IX,SP	DD 39
ADD IY,BC	FD 09
ADD IY,DE	FD 19
ADD IY,IY	FD 29
ADD IY,SP	FD 39
AND (HL)	A6
AND (IX+dis)	DD A6 XX
AND (IY+dis)	FD A6 XX
AND A	A7
AND B	A0
AND C	A1
AND D	A2
AND n	E6 XX
AND E	A3
AND H	A4
AND L	A5
BIT 0,(HL)	CB 46
BIT 0,(IX+dis)	DD CB XX 46
BIT 0,(IY+dis)	FD CB XX 46
BIT 0,A	CB 47
BIT o,B	CB 40
BIT 0,C	CB 41
BIT 0,D	CB 42
BIT 0,E	CB 43
BIT 0,H	CB 44
BIT 0,L	CB 45
BIT 1,(HL)	CB 4E
BIT 1,(IX+dis)	DD CB XX 4E
BIT 1,(IY+dis)	FD CB XX 4E
BIT 1,A	CB 4F
BIT 1,B	CB 48
BIT 1,C	CB 49
BIT 1,D	CB 4A
BIT 1,E	CB 4B
BIT 1,H	CB 4C
BIT 1,L	CB 4D
BIT 2,(HL)	CB 56
BIT 2,(IX+dis)	DD CB XX 56
BIT 2,(IY+dis)	FD CB XX 56
BIT 2,A	CB 57

MNEMONIC	HEXADECIMAL
BIT 2,B	CB 50
BIT 2,C	CB 51
BIT 2,D	CB 52
BIT 2,E	CB 53
BIT 2,H	CB 54
BIT 2,L	CB 55
BIT 3,(HL)	CB 5E
BIT 3,(IX+dis)	DD CB XX 5E
BIT 3,(IY+dis)	FD CB XX 5E
BIT 3,A	CB 5F
BIT 3,B	CB 58
BIT 3,C	CB 59
BIT 3,D	CB 5A
BIT 3,E	CB 5B
BIT 3,H	CB 5C
BIT 3,L	CB 5D
BIT 4,(HL)	CB 66
BIT 4,(IX+dis)	DD CB XX 66
BIT 4,(IY+dis)	FD CB XX 66
BIT 4,A	CB 67
BIT 4,B	CB 60
BIT 4,C	CB 61
BIT 4,D	CB 62
BIT 4,E	CB 63
BIT 4,H	CB 64
BIT 4,L	CB 65
BIT 5,(HL)	CB 6E
BIT 5,(IX+dis)	DD CB XX 6E
BIT 5,(IY+dis)	FD CB XX 6E
BIT 5,A	CB 6F
BIT 5,B	CB 68
BIT 5,C	CB 69
BIT 5,D	CB 6A
BIT 5,E	CB 6B
BIT 5,H	CB 6C
BIT 5,L	CB 6D
BIT 6,(HL)	CB 76
BIT 6,(IX+dis)	DD CB XX 76
BIT 6,(IY+dis)	FD CB XX 76
BIT 6,A	CB 77
BIT 6,B	CB 70
BIT 6,C	CB 71
BIT 6,D	CB 72
BIT 6,E	CB 73
BIT 6,H	CB 74
BIT 6,L	CB 75
BIT 7,(HL)	CB 7E
BIT 7,(IX+dis)	DD CB XX 7E
BIT 7,(IY+dis)	FD CB XX 7E
BIT 7,A	CB 7F
BIT 7,B	CB 78
BIT 7,C	CB 79
BIT 7,D	CB 7A
BIT 7,E	CB 7B
BIT 7,H	CB 7C
BIT 7,L	CB 7D
CALL ADDR	CD XX XX
CALL C,ADDR	DC XX XX
CALL M,ADDR	FC XX XX
CALL NC,ADDR	D4 XX XX
CALL NZ,ADDR	C4 XX XX
CALL P,ADDR	F4 XX XX
CALL PE,ADDR	EC XX XX
CALL PO,ADDR	E4 XX XX
CALL Z,ADDR	CC XX XX
CCF	3F
CP (HL)	BE
CP (IX+dis)	DD BE XX
CP (IY+dis)	FD BE XX
CP A	BF
CP B	B8
CP C	B9
CP D	BA

MNEMONIC	HEXADECIMAL
CP n	FE XX
CP E	BB
CP H	BC
CP L	BD
CPD	ED A9
CPDR	ED B9
CPI	ED A1
CPIR	ED B1
CPL	2F
DAA	27
DEC (HL)	35
DEC (IX+dis)	DD 35 XX
DEC (IY+dis)	FD 35 XX
DEC A	3D
DEC B	05
DEC BC	0B
DEC C	0D
DEC D	15
DEC DE	1B
DEC E	1D
DEC H	25
DEC HL	2B
DEC IX	DD 2B
DEC IY	FD 2B
DEC L	2D
DEC SP	3B
DI	F3
DJNZ,dis	10 XX
EI	FB
EX (SP) ,HL	E3
EX (SP) ,IX	DD E3
EX (SP) ,IY	FD E3
EX AF,AF'	08
EX DE,HL	EB
EXX	D9
HALT	76
IM 0	ED 46
IM 1	ED 56
IM 2	ED 5E
IN A, (C)	ED 78
IN A,port	DB XX
IN B, (C)	ED 40
IN C, (C)	ED 48
IN D, (C)	ED 50
IN E, (C)	ED 58
IN H, (C)	ED 60
IN L, (C)	ED 68
INC (HL)	34
INC (IX+dis)	DD 34 XX
INC (IY+dis)	FD 34 XX
INC A	3C
INC B	04
INC BC	03
INC C	0C
INC D	14
INC DE	13
INC E	1C
INC H	24
INC HL	23
INC IX	DD 23
INC IY	FD 23
INC L	2C
INC SP	33
IND	ED AA
INDR	ED BA
INI	ED A2
INIR	ED B2
JP (HL)	E9
JP (IX)	DD E9
JP (IY)	FD E9
JP ADDR	C3 XX XX
JP C,ADDR	DA XX XX
JP M,ADDR	FA XX XX

MNEMONIC	HEXADECIMAL
JP NC,ADDR	D2 XX XX
JP NZ,ADDR	C2 XX XX
JP P,ADDR	F2 XX XX
JP PE,ADDR	EA XX XX
JP PO,ADDR	E2 XX XX
JP Z,ADDR	CA XX XX
JR C,dis	38 XX
JR dis	18 XX
JR NC,dis	30 XX
JR NZ,dis	20 XX
JR Z,dis	28 XX
LD (ADDR),A	32 XX XX
LD(ADDR),BC	ED 43 XX XX
LD (ADDR),DE	ED 53 XX XX
LD(ADDR),HL	ED 63 XX XX
LD (ADDR),HL	22 XX XX
LD (ADDR),IX	DD 22 XX XX
LD (ADDR), IY	FD 22 XX XX
LD (ADDR),SP	ED 73 XX XX
LD (BC),A	02
LD (DE),A	12
LD (HL),A	77
LD (HL),B	70
LD (HL), C	71
LD (HL),D	72
LD (HL),n	36 XX
LD (HL),E	73
LD (HL),H	74
LD (HL),L	75
LD (IX+dis),A	DD 77 XX
LD (IX+dis),B	DD 70 XX
LD (IX+dis),C	DD 71 XX
LD (IX+dis),D	DD 72 XX
LD (IX+dis),n	DD 36 XX XX
LD (IX+dis),E	DD 73 XX
LD (IX+dis),H	DD 74 XX
LD (IX+dis),L	DD 75 XX
LD (IY+dis),A	FD 77 XX
LD (IY+dis),B	FD 70 XX
LD (IY+dis),C	FD 71 XX
LD (IY+dis),D	FD 72 XX
LD (IY+dis),n	FD 36 XX XX
LD (IY+dis),E	FD 73 XX
LD (IY+dis),H	FD 74 XX
LD (IY+dis),L	FD 75 XX
LD A,(ADDR)	3A XX XX
LD A,(BC)	0A
LD A,(DE)	1A
LD A,(HL)	7E
LD A,(IX+dis)	DD 7E XX
LD A,(IY+dis)	FD 7E XX
LD A,A	7F
LD A,B	78
LD A,C	79
LD A,D	7A
LD A,n	3E XX
LD A,E	7B
LD A,H	7C
LD A,I	ED 57
LD A,L	7D
LD A,R	ED 5F
LD B,(HL)	46
LD B,(IX+dis)	DD 46 XX
LD B,(IY+dis)	FD 46 XX
LD B,A	47
LD B,B	40
LD B,C	41
LD B,D	42
LD B,n	06 XX
LD B,E	43
LD B,H	44
LD B,L	45
LD BC,(ADDR)	ED 4B XX XX

MNEMONIC	HEXADECIMAL
LD BC,nn	01 XX XX
LD C,(HL)	4E
LD C,(IX+dis)	DD 4E xx
LD C,(IY+dis)	FD 4E XX
LD C,A	4F
LD C,B	48
LD C,C	49
LD C,D	4A
LD C,n	0E XX
LD C,E	4B
LD C,H	4C
LD C,L	4D
LD D,(HL)	56
LD D,(IX+dis)	DD 56 XX
LD D,(IY+dis)	FD 56 XX
LD D,A	57
LD D,B	50
LD D,C	51
LD D,D	52
LD D,n	16 XX
LD D,E	53
LD D,H	54
LD D,L	55
LD DE,(ADDR)	ED 5B XX XX
LD DE,nn	11 XX XX
LD E,(HL)	5E
LD E,(IX+dis)	DD 5E XX
LD E,(IY+dis)	FD 5E XX
LD E,A	5F
LD E,B	58
LD E,C	59
LD E,D	5A
LD E,n	1E XX
LD E,E	5B
LD E,H	5C
LD E,L	5D
LD H,(HL)	66
LD H,(IX+dis)	DD 66 XX
LD H,(IY+dis)	FD 66 XX
LD H,A	67
LD H,B	60
LD H,C	61
LD H,D	62
LD H,n	26 XX
LD H,E	63
LD H,H	64
LD H,L	65
LD HL,(ADDR)	ED 6B XX XX
LD HL,(ADDR)	2A XX XX
LD HL,nn	21 XX XX
LD I,A	ED 47
LD IX,(ADDR)	DD 2A XX XX
LD IX,nn	DD 21 XX XX
LD IY (ADDR)	FD 2A XX XX
LD IY,nn	FD 21 XX XX
LD L,A	6F
LD L,B	68
LD L,C	69
LD L,D	6A
LD L,n	2E XX
LD L,E	6B
LD L,(HL)	6E
LD L,(IX+dis)	DD 6E XX
LD L,(IY+dis)	FD 6E XX
LD L,H	6C
LD L,L	6D
LD R,A	ED 4F
LD SP,(ADDR)	ED 7B XX XX
LD SP,nn	31 XX XX
LD SP,HL	F9
LD SP,IX	DD F9
LD SP,IY	FD F9
LDD	ED A8

MNEMONIC	HEXADECIMAL
LDDR	ED B8
LDI	ED A0
LDIR	ED B0
NEG	ED 44
NOP	00
OR (HL)	B6
OR (IX+dis)	DD B6 XX
OR (IY+dis)	FD B6 xx
OR A	B7
OR B	B0
OR C	B1
OR D	B2
OR n	F6 XX
OR E	B3
OR H	B4
OR L	B5
OTDR	ED BB
OTIR	ED B3
OUT (C),A	ED 79
OUT (C),B	ED 41
OUT (C),C	ED 49
OUT (C),D	ED 51
OUT (C),E	ED 59
OUT (C),H	ED 61
OUT (C),L	ED 69
OUT port,A	D3 port
OUTD	ED AB
OUTI	ED A3
POP AF	F1
POP BC	C1
POP DE	D1
POP HL	E1
POP IX	DD E1
POP IY	FD E1
PUSH AF	F5
PUSH BC	C5
PUSH DE	D5
PUSH HL	E5
PUSH IX	DD E5
PUSH IY	FD E5
RES 0,(HL)	CB 86
RES 0,(IX+dis)	DD CB XX 86
RES 0,(IY+dis)	FD CB XX 86
RES 0,A	CB 87
RES 0,B	CB 80
RES 0,C	CB 81
RES 0,D	CB 82
RES 0,E	CB 83
RES 0,H	CB 84
RES 0,L	CB 85
RES 1,(HL)	CB 8E
RES 1,(IX+dis)	DD CB XX 8E
RES 1,(IY+dis)	FD CB XX 8E
RES 1,A	CB 8F
RES 1,B	CB 88
RES 1,C	CB 89
RES 1,D	CB 8A
RES 1,E	CB 8B
RES 1,H	CB 8C
RES 1,L	CB 8D
RES 2,(HL)	CB 96
RES 2,(IX+dis)	DD CB XX 96
RES 2,(IY+dis)	FD CB XX 96
RES 2,A	CB 97
RES 2,B	CB 90
RES 2,C	CB 91
RES 2,D	CB 92
RES 2,E	CB 93
RES 2,H	CB 94
RES 2,L	CB 95
RES 3,(HL)	CB 9E
RES 3,(IX+dis)	DD CB XX 9E
RES 3,(IY+dis)	FD CB XX 9E
RES 3,A	CB 9F

MNEMONIC	HEXADECIMAL
RES 3,B	CB 98
RES 3,C	CB 99
RES 3,D	CB 9A
RES 3,E	CB 9B
RES 3,H	CB 9C
RES 3,L	CB 9D
RES 4, (HL)	CB A6
RES 4, (IX+dis)	DD CB XX A6
RES 4, (IY+dis)	FD CB XX A6
RES 4,A	CB A7
RES 4,B	CB A0
RES 4,C	CB A1
RES 4,D	CB A2
RES 4,E	CB A3
RES 4,H	CB A4
RES 4,L	CB A5
RES 5 (HL)	CB AE
RES 5, (IX+dis)	DD CB XX AE
RES 5, (IY+dis)	FD CB XX AE
RES 5,A	CB AF
RES 5,B	CB A8
RES 5,C	CB A9
RES 5,D	CB AA
RES 5,E	CB AB
RES 5,H	CB AC
RES 5,L	CB AD
RES 6, (HL)	CB B6
RES 6, (IX+dis)	DD CB XX B6
RES 6, (IY+dis)	FD CB XX B6
RES 6,A	CB B7
RES 6,B	CB B0
RES 6,C	CB B1
RES 6,D	CB B2
RES 6,E	CB B3
RES 6,H	CB B4
RES 6,L	CB B5
RES 7, (HL)	CB BE
RES 7, (IX+dis)	DD CB XX BE
RES 7, (IY+dis)	FD CB XX BE
RES 7,A	CB BF
RES 7,B	CB B8
RES 7,C	CB B9
RES 7,D	CB BA
RES 7,E	CB BB
RES 7,H	CB BC
RES 7,L	CB BD
RET	C9
RET C	D8
RET M	F8
RET NC	D0
RET NZ	C0
RET P	F0
RET PE	E8
RET PO	E0
RET Z	C8
RETI	ED 4D
RETN	ED 45
RL (HL)	CB 16
RL (IX+dis)	DD CB XX 16
RL (IY+dis)	FD CB XX 16
RL A	CB 17
RL B	CB 10
RL C	CB 11
RL D	CB 12
RL E	CB 13
RL H	CB 14
RL L	CB 15
RLA	17
RLC (HL)	CB 06
RLC (IX+dis)	DD CB XX 06
RLC (IY+dis)	FD CB XX 06
RLC A	CB 07
RLC B	CB 00
RLC C	CB 01
RLC D	CB 02
RLC E	CB 03
RLC H	CB 04
RLC L	CB 05
RLCA	07
RLD	ED 6F
RR (HL)	CB 1E
RR (IX+dis)	DD CB XX 1E
RR (IY+dis)	FD CB XX 1E
RR A	CB 1F
RR B	CB 18
RR C	CB 19
RR D	CB 1A
RR E	CB 1B
RR H	CB 1C
RR L	CB 1D
RRA	1F
RRC (HL)	CB 0E
RRC (IX+dis)	DD CB XX 0E
RRC (IY+dis)	FD CB XX 0E
RRC A	CB 0F
RRC B	CB 08
RRC C	CB 09
RRC D	CB 0A
RRC E	CH 0B
RRC H	CB 0C
RRC L	CB 0D
RRCA	0F
RRD	ED 67
RST 00	C7
RST 08	CF
RST 10	D7
RST 18	DF
RST 20	E7
RST 28	EF
RST 30	F7
RST 38	FF
SBC A, (HL)	9E
SBC A, (IX+dis)	DD 9E XX
SBC A, (IY+dis)	FD 9E XX
SBC A,A	9F
SBC A,B	98
SBC A,C	99
SBC A,D	9A
SBC A,n	DE XX
SBC A,E	9B
SBC A,H	9C
SBC A,L	9D
SBC HL,BC	ED 42
SBC HL,DE	ED 52
SBC HL,HL	ED 62
SBC HL,SP	ED 72
SCF	37
SET 0, (HL)	CB C6
SET 0, (IX+dis)	DD CB XX C6
SET 0, (IY+dis)	FD CB XX C6
SET 0,A	CB C7
SET 0,B	CB C0
SET 0,C	CB C1
SET 0,D	CB C2
SET 0,E	CB C3
SET 0,H	CB C4
SET 0,L	CB C5
SET 1, (HL)	CB CE
SET 1, (IX+dis)	DD CB XX CE
SET 1, (IY+dis)	FD CB XX CE
SET 1,A	CB CF
SET 1,B	CB C8
SET 1,C	CB C9
SET 1,D	CB CA
SET 1,E	CB CB
SET 1,H	CB CC
SET 1,L	CB CD
SET 2, (HL)	CB D6
SET 2, (IX+dis)	DD CB XX D6
SET 2, (IY+dis)	FD CB XX D6
SET 2,A	CB D7
SET 2,B	CB D0
SET 2,C	CB D1
SET 2,D	CB D2
SET 2,E	CB D3
SET 2,H	CB D4
SET 2,L	CB D5
SET 3, (HL)	CB DE
SET 3, (IX+dis)	DD CB XX DE
SET 3, (IY+dis)	FD CB XX DE
SET 3,A	CB DF
SET 3,B	CB D8
SET 3,C	CB D9
SET 3,D	CB DA
SET 3,E	CB DB
SET 3,H	CB DC
SET 3,L	CB DD
SET 4, (HL)	CBE6
SET 4, (IX+dis)	DD CB XX E6
SET 4, (IY+dis)	FD CB XX E6
SET 4,A	CB E7
SET 4,B	CB E0
SET 4,C	CB E1
SET 4,D	CB E2
SET 4,E	CB E3
SET 4,H	CB E4
SET 4,L	CB E5
SET 5, (HL)	CB EE
SET 5, (IX+dis)	DD CB XX EE
SET 5, (IY+dis)	FD CB XX EE
SET 5,A	CB EF
SET 5,B	CB E8
SET 5,C	CB E9
SET 5,D	CB EA
SET 5,E	CB EB
SET 5,H	CB EC
SET 5,L	CB ED
SET 6, (HL)	CB F6
SET 6, (IX+dis)	DD CB XX F6
SET 6, (IY+dis)	FD CB XX F6
SET 6,A	CB F7
SET 6,B	CB F0
SET 6,C	CB F1
SET 6,D	CB F2
SET 6,E	CB F3
SET 6,H	CB F4
SET 6,L	CB F5
SET 7, (HL)	CB FE
SET 7, (IX+dis)	DD CB XX FE
SET 7,(IY+dis)	FD CB XX FE
SET 7,A	CB FF
SET 7,B	CB F8
SET 7,C	CB F9
SET 7,D	CB FA
SET 7,E	CB FB
SET 7,H	CB FC
SET 7,L	CB FD
SLA (HL)	CB 26
SLA (IX+dis)	DD CB XX 26
SLA (IY+dis)	FD CB XX 26
SLA A	CB 27
SLA B	CB 20
SLA C	CB 21
SLA D	CB 22
SLA E	CB 23
SLA H	CB 24
SLA L	CB 25
SRA (HL)	CB 2E
SRA (IX+dis)	DD CB XX 2E

MNEMONIC	HEXADECIMAL	MNEMONIC	HEXADECIMAL	MNEMONIC	HEXADECIMAL
SRA (IY+dis)	FD CB XX 2E				
SRA A	CB 2F				
SRA B	CB 28				
SRA C	CB 29				
SRA D	CB 2A				
SRA E	CB 2B				
SRA H	CB 2C				
SRA L	CB 2D				
SRL (HL)	CB 3E				
SRL (IX+dis)	DD CB XX 3E				
SRL (IY+dis)	FD CB XX 3E				
SRL A	CB 3F				
SRL B	CB 38				
SRL C	CB 39				
SRL D	CB 3A				
SRL E	CB 3B				
SRL H	CB 3C				
SRL L	CB 3D				
SUB (HL)	96				
SUB (IX+dis)	DD 96 XX				
SUB (IY+dis)	FD 96 XX				
SUB A	97				
SUB B	90				
SUB C	91				
SUB D	92				
SUB E	93				
SUB n	D6 XX				
SUB H	94				
SUB L	95				
XOR (HL)	AE				
XOR (IX+dis)	DD AE XX				
XOR (IY+dis)	FD AE XX				
XOR A	AF				
XOR B	A8				
XOR C	A9				
XOR D	AA				
XOR n	EE XX				
XOR E	AB				
XSOR H	AC				
XOR L	AD				

Appendix 3.

Flag Operation Summary

INSTRUCTION	C	Z	P/V	S	N	H	COMMENTS
ADC HL, SS	#	#	V	#	0	X	16-bit add with carry
ADX s; ADD s	#	#	V	#	0	#	8-bit add or add with carry
ADD DD, SS	#	–	–	–	0	X	16-bit add
AND s	0	#	P	#	0	1	Logical operations
BIT b, s	–	#	X	X	0	1	State of bit b of location s is copied into the Z flag
CCF	#	–	–	–	0	X	Complement carry
CPD; CPDR; CPI; CPIR	–	#	#	X	1	X	Block search instruction Z=1 if A=(HL), else Z=0 P/V=1 if BC≠0, otherwise P/V=0
CP s	#	#	V	#	1	#	Compare accumulator
CPL	–	–	–	–	1	1	Complement accumulator
DAA	#	#	P	#	–	#	Decimal adjust accumulator
DEC s	–	#	V	#	1	#	8-bit decrement
IN r, (C)	–	#	P	#	0	0	Input register indirect
INC s	–	#	V	#	0	#	8-bit increment
IND; INI	–	#	X	X	1	X	Block input Z=0 if B≠0 else Z=1
INDR:INIR	–	1	X	X	1	X	Block input Z=0 if B≠0 else Z=1
LD A,I ; LD A,R	–	#	IFF	#	0	0	Content of interrupt enable Flip-Flop is copied into the P/V flag
LDD; LDI	–	X	#	X	0	0	Block transfer instructions
LDDR; LDIR	–	X	0	X	0	0	P/V=1 if BC≠0, otherwise P/V=0
NEG	#	#	V	#	1	#	Negate accumulator
OR s	0	#	P	#	0	0	Logical OR accumulator
OTDR; OTIR	–	1	X	X	1	X	Block output; Z=0 if B≠0 otherwise Z=1
OUTD; OUTI	–	#	X	X	1	X	Block output; Z=0 if B≠0 otherwise Z=1
RLA; RLCA; RRA; RRCA	#	–	–	–	0	0	Rotate accumulator
RLD; RRD	–	#	P	#	0	/	Rotate digit left and right
RLS; RLC s; RR s; RRC s SLA s; SRA s; SRL s	#	#	P	#	0	0	Rotate and shift location s
SBC HL, SS	#	#	V	#	1	X	16-bit subtract with carry
SCF	1	–	–	–	0	0	Set carry
SBC s; SUB s			V		1		8-bit subtract with carry
XOR x	0		P		0	0	Exclusive OR accumulator

SYMBOL	OPERATION
C	Carry flag. C =1 if the operation produced a carry from the most significant bit of the operand or result.
Z	Zero flag. Z =1 if the result of the operation is zero.
S	Sign flag. S =1 if the most significant bit of the result is one, i.e. a negative number.
P/V	Parity or overflow flag. Parity (P) and overflow (Ø) share the same flag. Logical operations affect this flag with the parity of the result while arithmetic operations affect this flag with the overflow of the result. If P/V holds parity, P/V =1 if the result of the operation is even, P/V =Ø if result is odd. If P/V holds overflow, P/V =1 if the result of the operation produced an overflow.
H	Half-carry flag. H =1 if the add or subtract operation produced a carry into or borrow from bit 4 of the accumulator.
N	Add/Subtract flag. N =1 if the previous operation was a subtract. H and N flags are used in conjunction with the decimal adjust instruction (DAA) to properly correct the result into packed BCD format following addition or subtraction using operands with packed BCD format.
#	The flag is affected according to the result of the operation.
—	The flag is unchanged by the operation.
Ø	The flag is reset (=Ø) by the operation.
1	The flag is set (=1) by the operation.
X	The flag result is unknown.
V	The P/V flag is affected according to the overflow result of the operation.
P	P/V flag is affected according to the parity result of the operation.
r	Any one of the CPU registers A,B,C,D,E,H,L.
s	Any 8-bit location for all the addressing modes allowed for the particular instructions.
SS	Any 16-bit location for all the addressing modes allowed for that instruction.
R	Refresh register.
n	8-bit value in range Ø-255.
nn	16-bit value in range Ø-65535.

NOTES

NOTES

NOTES

NOTES

NOTES

Give your Amstrad programs the power and speed of machine language without actually having to learn machine language programming.

Now, without any additional effort, you can overcome the limitations of BASIC. The routines in this book will help you to develop programs of professional quality; not only will your programs look better by giving you direct access and control over all the graphics features, run faster, and be more spectacular — with superb sound effects — but the development time will be a fraction of what you would normally expect.

The routines in this book are all presented in a format that is both easy to enter and understand. They also have accompanying notes and requirements, so that alteration or enhancement becomes a simple task.

The book includes routines such as displaying large characters, manipulating and displaying strings, inverting characters and screen, displaying graphics shapes, drawing and much more.

Whether you are a beginner Amstrad CPC 464/CPC 664 user or an experienced programmer, this is a book you should have by your side at all times.

£7.95

Melbourne
House
Publishers

ISBN 0-86161-198-5

9 780861 611980

Ready Made Machine Language Routines for the Amstrad