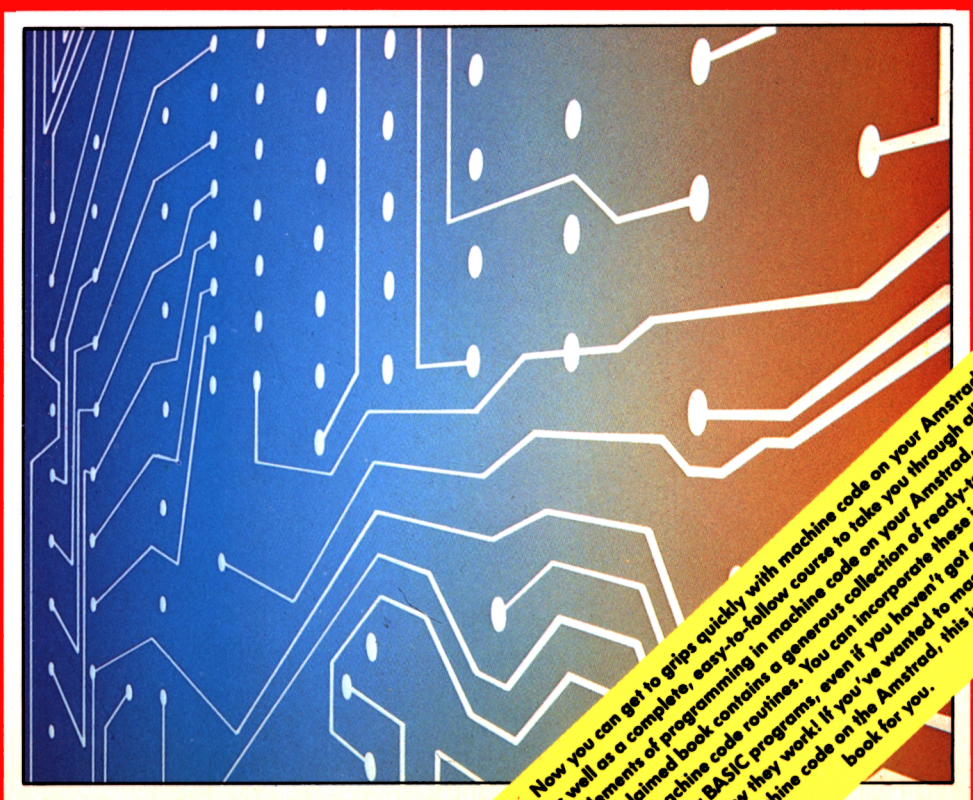
MASTERING MACHINE CODE ON YOUR AMSTRAD

464/664/6128

Clive Gifford & Scott Vincent



Now you can get to grips quickly with machine code on your Amstrad. As well as a complete, easy-to-follow course to take you through all the elements of programming in machine code on your Amstrad, this acclaimed book contains a generous collection of ready-to-run machine code routines. You can incorporate these into your BASIC programs, even if you haven't got a clue how they work! If you've wanted to master machine code on the Amstrad, this is the book for you.

Interface Publications
LONDON · MELBOURNE

MASTERING MACHINE CODE ON YOUR AMSTRAD 464/664/6128

CLIVE GIFFORD AND
SCOTT VINCENT

Best Wishes,
C. Gifford

MASTERING MACHINE CODE ON YOUR AMSTRAD 464/664/6128

Dedication:

To all of our friends at Sheffield and Southampton Universities.

Acknowledgements:

To Sally and Peter for their enormous help with this book; to Andy, Doug and Luke for starting the decorating; to Catherine for her letters; and to summer opening hours for providing our main inspiration.

First published in the UK by:
Interface Publications Ltd.,
9-11 Kensington High Street,
London W8 5NP

First published in Australia by:
Interface Publications (Australia) Pty. Ltd.,
Chelsea House,
34 Camp Street,
Chelsea, Victoria, 3196

First printing March, 1986

ISBN 0 907563 91 0

Books published by Interface Publications are distributed in the UK by WHSD, St John's House, East Street, Leicester LE1 6NE (0533 551196) and in Australia and New Zealand by Pitman Publishing (Melbourne, Sydney, Brisbane and Wellington). Any queries regarding the contents of this volume should be directed by mail to Interface Publications at either the London or Chelsea, Australia, address.

Typeset by Twickenham Printers
Cover photograph – Tony Stone Associates

Contents

Pre-Packaged Machine Code Routines:

Foreword – Tim Hartnell

Now you have the chance to really master machine code programming on your Amstrad computer. No matter what anybody says, it is not an easy subject to come to grips with.

However, with the right guidance, even the most difficult terrain can be negotiated. I believe that Clive and Scott – two highly competent programmers, with a wealth of software and book writing experience behind them – are the ideal guides to help you understand the intricacies of Amstrad machine code programming.

You should work right through the book, skipping those sections which seem difficult the first time you read it. By the time you've finished the first reading, you should know enough to make sense of the sections whose meaning eluded you the first time through.

If you do this, you'll find the task is made much easier, and your progress won't be impeded because one particular section seems harder than the others.

And don't forget all the 'ready-to-run' machine code routines in this book (including a complete machine code games writing package) which you can use, in conjunction with your BASIC programs, even if you don't understand – at this stage – how they work.

Time now to start mastering machine code on your Amstrad.

Tim Hartnell,
London, 1986

Tim Hartnell, head of Interface Publications, is one of the most widely-published computer authors in the world. Recent works include HARNESSING THE POWER OF YOUR ATARI ST, REPLICATING REALITY: EXPLORING COMPUTER SIMULATIONS and EXPLORING EXPERT SYSTEMS ON YOUR MICROCOMPUTER.

Authors' Introduction

We hope that this book helps you master programming in machine code. We've tried to write an easy-to-follow tutorial to give an insight into the workings of machine code, and its main commands, as well as providing a host of useful charts and tables. These take up about half of the book. The balance is a generous selection of machine code routines. These routines include an easy-to-use BASIC loader that can be typed in and run in a few minutes. And even if you don't want to learn the intricacies of machine code right now, this book will give you a library of useful and (in some cases) amazing routines.

It's been hard but enjoyable work writing this book. We hope you get as much value from the book as we did writing it.

Scott Vincent
Clive Gifford

London, March 1986

Scott Vincent is a very experienced machine code programmer. Currently studying computer science at Sheffield University, Scott has had several machine code games published by well-known software houses. He has written many books which have been published by Interface and Virgin Books.

Clive Gifford has over twenty books in print and writes frequently for a number of computing magazines. His published titles include 'The Amstrad Programmer's Reference Guide', 'Using Computers In Education' and 'Using a Modem With Your Computer'.

The authors have worked previously on a number of projects together including several of the 'Games For Your...' series of books published by Virgin.

Chapter One
What is Machine Code?

Firstly, machine code is not some magical code that instantly creates programs of the same standard as commercial software. It is slow and laborious to write, very difficult to debug and often produces results that are only marginally better than a similar routine written in BASIC. It is not a wonder language and we feel that too much esteem is given to the terms '100% machine code program' and 'machine code programmer'.

To understand how machine code works, we need first to have a brief look into the internal workings of your computer. Your computer's processor consists of thousands of "gates", or switches which are either open (off) or closed (on). Binary numbers can be used to represent these settings easily with a one corresponding to on, and a zero for off. However, entering hundreds of ones and noughts just to get one command working is not an efficient use of time and so "high level" computer languages were created. BASIC is one such language. It replaces the masses of ones and zeros with easy-to-understand, English-like commands. It's not that difficult to ascertain what PRINT, CLEAR and END do. An *interpreter* is needed to translate BASIC commands into binary digits which the computer can understand, and this interpreting slows the speed of operation down greatly.

Machine code is the 'machine-level' series of instructions into which BASIC is converted. These low level instructions can then be acted upon directly by the processor.

The 'brains' of your computer, where much of the action occurs, is the Central Processing Unit or **CPU.** Each design of CPU has its own collection of low-level machine instructions, known as the **Instruction Set.**

The CPU in your Amstrad is one of the most popular designs found in microcomputers, the Zilog Z8Ø CPU chip. So, naturally enough, this book deals with how to program in Z8Ø machine code and how to use the Z8Ø Instruction Set. (A number of other computers use the Z8Ø chip, the Spectrum and the MSX series being just two machines. But on each machine, there are certain conditions so one finds that while the Z8Ø chip is in a range of home computers, actually using it is a little different with each machine. A good example with the Amstrad is the non-standard way in which the screen is handled.)

The Instruction Set

Each machine code instruction forms part of the low level language. It may take half a dozen or more machine code instructions to emulate a single BASIC command.

There are a number of ways of representing machine code. One such way is **Assembly Language** which involves allocating a short descriptive name, known as a **mnemonic,** to each instruction. Another way is to use a decimal number. Binary and hexadecimal (base 16) numbers can also be used (more of that in the next chapter). For example, the instruction which makes the computer return from the machine code level to BASIC has a decimal value of 2Ø1, a binary value of 11ØØ1ØØ1, a hexadecimal value of C9 and a mnemonic of RET.

Now we have a very brief idea of what machine code is, let's look at the reasons for and against learning how to use it.

In Favour

Machine code, when written well, is usually much faster than a comparable BASIC routine.

Machine code allows you to perform actions that are impossible in BASIC, such as speech synthesis. As well machine code often takes far less memory than a similar BASIC program.

As well as this, there is a great deal of prestige in being able to program in machine code. Programs which work perfectly well in BASIC are often chastised in reviews because they are not written in machine code even when writing it in machine code would not produce any improvement whatsoever.

Arguments Against

Machine code often needs many instructions to emulate a simple action.

Machine code tends to be difficult to read, understand and debug. Writing machine code programs is often more difficult and time-consuming than writing in a high level language such as BASIC.

It is very difficult to perform anything more than simple arithmetic in machine code, and the code can be very difficult to transfer from machine to machine. It is difficult enough if you are transferring a Z8Ø machine code program to another Z8Ø-based machine, but attempting to transfer a Z8Ø routine to, say, a Commodore 64 (which uses a 65Ø2 CPU) is practically impossible.

Chapter Two
Number Systems and Assembly Language

The three number systems that are often used when using machine code are decimal, binary and hexadecimal.

As you know, decimal numbers are built on a base of ten, binary numbers on a base of two, and hexadecimal numbers on a base of sixteen.

Imagine that, inside your Amstrad's CPU, are a whole stack of little pigeon holes, called addresses. Each address can hold eight binary digits (and a group of eight binary digits is known, for some arcane reason, as a **byte**). Obviously, if all eight digits are zeros, the number held in the address is zero. If, however, all eight digits are ones (i.e. 11111111), the address holds the number 255. Therefore, an address can hold any number between zero and 255.

The Amstrad command BIN$ converts decimal numbers into their binary equivalents:

BIN$(V,N)

In this statement, V is the value in decimal to be altered and N is the number of digits displayed. (The last parameter needn't be used, in which case the Amstrad will print the result without any leading

zeros. PRINT BIN$(4) will return an answer of 1ØØ, whereas if you specify that you want all of the eight digits you will get the answer ØØØØØ1ØØ.)

Hexadecimal (or "hex") is the correct name for the numbering system based on sixteen. The range of digits is Ø, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. Hexadecimal is used in machine code because it is closely related to, and easy to convert to, binary, the computer's natural numbering system. Now, a single hexadecimal digit equals four binary digits (bits) or half a byte (a byte is eight bits). Half a byte is called a nibble! Thus Ø in hex equals ØØØØ in binary, 2 equals ØØ1Ø and F equals 1111. PRINT HEX$(2Ø1) will produce the answer C9. This is obtained by 16 times the decimal value of the first hex digit (C=12) plus the decimal value of the second hex digit (9=9), or 192+9 which equals 2Ø1.

If you wish to include a hex number in a program, it should be prefixed with either "&" or "&H". A Binary number is prefixed with "&X".

Sixteen and Eight-Bit Numbers

So far, we have dealt with numbers in the decimal range of zero to 255. These are, as we said earlier, 8-bit numbers. However, to get numbers greater than 255, we need two byte (16-bit) numbers. These have a far larger range, between Ø and 65535. If you had a 16-bit number shown in hex as Ø3C9 then the Ø3 is known as the **most significant** or **high** byte while the other byte, C9, is the **low** or **least significant** byte. The total value of a 16-bit number is calculated as:

256 times high byte plus low byte

256 * Ø3 + C9 = 256 x 3 + 2Ø1 = 969

So we have seen how positive numbers between Ø and 65535 are obtained, but what about negative numbers? You have a most significant and least significant **bit** within a **byte.** The most significant bit is the **leftmost one,** and the furthest right is the least significant bit. Taking this **most significant bit** (which is normally worth either Ø or 128) we use **it** as the **sign** of the number. This effectively cuts down the maximum size of an 8-bit number to 127. However, it keeps the **range** of 256, as now you can have any value between -128 and +127. Setting this bit to zero makes the rest of the byte a **positive** value, while setting it to one will give a negative value. 16-bit numbers work in a similar way with the leftmost bit of the most significant or high byte becoming the **sign bit** as it is called.

However, creating negative binary numbers is unfortunately not quite as simple as that. The rule that adding the corresponding positive number to the negative value should result in the answer zero must be followed. To obtain this, we must use a number concept called **Two's Complement.** This is a two-part process. Firstly, you must swap the value of each bit of the byte. Therefore if a bit was equal to 1 then it should be made equal to Ø and vice versa. This is the actual complementing process. After this has been done, you add 1 to the result and this will equal the opposite-signed binary value of a number.

Hopefully, we can make this a little clearer with an example:

To change +7 to −7:

+7 = ØØØØØ111

Complement the number = 11111ØØØ

Add one to the complement = 11111ØØ1

Therefore, 11111ØØ1 is the binary equivalent of −7.

(This works equally as well for conversions from minus to plus. You will notice that using this system, the signed bit is set correctly.

16-bit numbers are calculated in the same way and Amstrad's Basic holds all integer variables as 16-bit two's complement numbers. As with 8-bit numbers the range is kept the same but the maximum number is halved to 32767.)

Assembly Language

We hope you are still with us. Unfortunately, we need to go through this before getting to the interesting bits. Stick with it. It will be well worth it.

First, here's another coding system which you need to know about. **Assembly language** offers an alternative to using decimal and hex numbers to represent machine code routines and instructions. Whereas an instruction is shown as a number in decimal or hex, in assembly language it is shown by a mnemonic, a short name. A list of numbers would be very difficult to follow but a list of mnemonics such as RET (short for RETURN), JP (short for JUMP) and LD (short for LOAD) is far easier to understand.

In the back of the book there is an appendix which gives all of the Z8Ø instructions. These are displayed in both mnemonic and hexadecimal forms. As well, to make it easier to follow what is going on, all of the machine code routines in this book are supplied in both BASIC and assembly language forms. The BASIC listing is the one that you are most likely to actually use as this can be typed straight into your Amstrad and run. The assembly language listing needs a special program (called an **assembler**) to convert the mnemonics into code that can be acted upon by the Amstrad.

Buying an Assembler

There are many assemblers on the market. While they all offer varied additional features, all perform the one essential task of

turning assembly language listings into proper code. The value and importance of an assembler cannot be underestimated. You would find it extremely difficult to write your first machine code programs in decimal or hex, or converting the mnemonics into hex by hand, without an assembler. As you progress in the machine code world, you will find that longer routines are much simpler to write on an assembler. If you pick the right one, you'll also get a host of very useful features to aid writing, debugging and storing your machine code routines.

Despite this, you do not need an assembler to proceed with the material in this book. If you are serious about machine code, though, you will want to consider the purchase of one fairly soon.

If you are committed to machine code as something you wish to learn and use then you might as well buy a full-feature assembler, preferably one that comes with a monitor (a program allowing you to scan through memory) as well. Good ones available include **The Code Machine** by **Picturesque Software** costs a hefty £19.95 at the time of writing but well worth the money. We're not going to go into a long discussion of all the features save to say this is the assembler that we used in the creation of this book. Picturesque can be contacted at 6, Corkscrew Hill, West Wickham, Kent BR4 9BB.

If something a little cheaper is more to your taste then why not have a look at the **Amstrad Assembly Language Course** supplied by **Glentop Publishers** of Barnet, Herts. The assembler supplied with the course is pretty good, and as well you get a machine code course, all for £12.95 at the time of writing. Glentop have a freepost address, so it will cost you nothing to get some more information.

If you are an impecunious programmer, you can always turn to *'Computing With The Amstrad'* magazine. Issue number 7 (July, 1985) included a reasonable assembler listing. Back issues of the magazines can be obtained from Freepost, Europa House, 68

Chester Road, Hazel Grove, Stockport SK7 5NY. Each back issue costs around £1.25 while a tape containing all the programs in the magazine costs £3.75 or so.

Directives

Moving on from assembler choice, we need to mention the various assembler **directives.** These are commands that you will find in the assembly language listing that are not actual machine code instructions but are commands produced by the assembler to aid the machine code programmer.

These vary slightly from assembler to assembler but usually include the main ones detailed below:

ORG – defines the address of the first piece of assembled code.

END – signifies the end of the program.

EQU – assigns a value to a label name.

DEFL – allows you to give a label a value as often as you like.

DEFB – assigns a value to a single byte at the current assembly address.

DEFW – assigns a value to the next two bytes at the current assembly address (a double-byte version of DEFB).

DEFS – creates a number of blank bytes from the current assembly address onwards.

DEFM - allows messages to be entered as a string of characters.

PRNT - allows printer to be turned on and off while the assembler is running through the listing attempting to assemble the code. The exact nature of this command varies depending on the assembler.

ENT - defines the entry point of the program.

Don't worry if you do not fully understand all of these directives. When you start using an assembler or are looking at the assembly language listings, you can always turn back to these pages.

Chapter Three
Using Machine Code on the Amstrad

In this chapter, we'll look at the Amstrad's hardware, how to prepare the machine for a machine code program and how to save and load the program.

The Hardware

Let's have a brief look at the various parts that make up an Amstrad computer. These parts can be divided into input, output and processing. Input obviously includes the keyboard, but it can also take in such things as a joystick and any other input device such as a mouse.

Output includes the monitor and any interfaces which allow data to be output to a printer or another external device.

The internal processing parts of the Amstrad are most important for our discussion on machine code. As mentioned in the first chapter, the Amstrad is based around the Z8Ø CPU. The CPU is the central 'thinking' unit of the computer which co-ordinates the workings of all the other parts. Memory is perhaps the most essential of all the other parts.

Memory comes in two basic types, ROM and RAM. ROM is Read

Only Memory and cannot be altered through machine code. Its impregnability is needed as it holds important information - the operating system and the BASIC interpreter. As its name suggests, you can look at the ROM values, which can be very useful as we'll later see.

RAM stands for Random Access Memory and is the type of memory that you use to store your programs in. Not only can it be read, it can also be altered.

The Amstrad consists of 65536 RAM locations and 32768 ROM locations. You may be more familiar with the terms 64K and 32K (a K being a term for 1Ø24). Each location is capable of holding an 8-bit number which as we learned from binary in the previous chapter, means a number in the decimal range, Ø to 255. We also know that this 8-bit memory location is called a byte.

With that out of the way, let's turn to the other parts of the Amstrad.

The Sound

The PSG chip is the Programmable Sound Generator of the Amstrad. It is an AY-3-8912 which is found in many other home computers. The PPI is the Programmable Peripheral Interface and this important chip is the link between the CPU and the PSG, the printer port, the screen display and the cassette and/or disc drive. The CRTC is also concerned with the screen display (its full name is the Cathode Ray Tube Controller).

The final piece of the Amstrad's hardware to take a look at is the Gate Array. This is a nifty device, specific only to the Amstrad, which helps overlap the ROM and RAM as well as having a hand in the generation of the screen display. The overlapping of ROM and RAM will be discussed, along with the memory map, later.

Writing a Machine Code Program

Moving on from the Amstrad's hardware, we'll now look at the procedures involved in writing a machine code program.

Firstly, we must **reserve** an area of **memory** to place the machine code program in. Although your Amstrad has 64K (or more if you have the 6128 machine), not all of this memory is available for your programs. If you type PRINT FRE(Ø), you will be told that there is over 42K of memory at your disposal. However, you will only need a fraction of that for most machine code programs.

This spare user memory has an upper boundary above which lies the code for controlling the computer. This upper limit is called HIMEM and is a variable that can be printed and altered. If you switch your machine on and enter PRINT HIMEM, you should get a result of around 43K depending on which Amstrad model you have. HIMEM can be altered to a value below this original upper limit, via the MEMORY command. MEMORY 4ØØØØ lowers HIMEM to the address 4ØØØØ and gives you several K of RAM in which to place your machine code program or routine. We have done this with all the routines in the latter half of the book. The area above HIMEM is not affected by the command NEW.

Symbol After

There is another point concerning memory reservation that we must cover. The SYMBOL AFTER command reserves memory for using user-defined graphics. This memory is reserved immediately below HIMEM and stays there even if HIMEM is altered. If, for example, HIMEM was first 4ØØØØ and then altered to 36ØØØ, the memory for the UDG's would be above HIMEM and would not be accessed by the computer. This would cause many problems.

If you turn to one of the many routines in the second half of the book, you will see that the first program line usually reads:

1Ø SYMBOL AFTER 256:MEMORY 39999:SYMBOL AFTER 24Ø

This line first gets rid of all memory saved for UDG's, alters HIMEM down to 39999 (meaning the first available address for a machine code routine is 4ØØØØ, one above that) and then reserves memory for 16 UDG's starting from the new HIMEM position downwards. (For a more detailed explanation of SYMBOL AFTER and user-defined graphics, refer to the Amstrad manual.)

Once you have your machine code program entered into memory save it AT ONCE. Whatever you do, don't RUN the program without having a copy of it on tape or disc. Unlike BASIC programs with their friendly error commands, if a machine code program doesn't work properly, it is more than likely that it will 'crash', leaving you to switch your computer off and on again, thus losing all your hard work. We really cannot stress this point too strongly.

Saving It All

Saving a machine code program is just a matter of adding several extra parameters to the normal SAVE command. The new format is:

SAVE "PROGNAME", B, start address, length, entry point

The start address is the memory address from which program will be saved, while the entry point is the address from which the program actually runs.

The length of the program is the number of bytes that the routine occupies. This can be calculated by the following formula: End Address Of Routine – Start Address Of Routine + 1.

Loading It In

Loading a machine code program is simplicity itself. You just type LOAD "PROGNAME" or even LOAD "". You can re-locate the program by specifying a new start address, e.g. LOAD "SCREEN", 30000. However, you must bear in mind that the program may contain instructions that are dependant upon the original start address (this will become clearer in the following chapters, particularly the chapter concerned with the jump instructions).

Chapter Four
Your First Machine Code Program

You now have a pretty good grounding in how to prepare for a machine code routine. As well, you now know what assembly language is. Now has come the time to use this in your first machine code program.

Recently, we read a letter in an Amstrad-specific magazine from a puzzled reader saying that they couldn't get their machine code working. They typed in hex pairs of numbers such as C9, and mnemonics such as RET and LD straight into the machine. Of course, all they got for their trouble was a series of 'syntax error' messages.

We already know that this process would not work as we know that (a) for using mnemonics we need an assembler and (b) code has to be actually POKEd into the memory of your Amstrad.

A Hex Loader

To do this job you need a small program known as a loader. Routines are often presented with an integral loader, such as all the routines in the second half of the book. For very short routines, such as the ones that we will be dealing with over the course of the next few chapters, it is better to use a special hexloading program. These

are generally simple affairs (though we have seen some that add so many fancy features that they amount to pages of pages of listing just to perform the same function as a ten-line program).

Type in the following program and save it on disc or tape. You will need it a number of times before this book is out.

```
1 '           HEX LOADER
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 240
20 n=40000
30 INPUT a$
40 IF LEN(A$)/2<>INT(LEN(A$)/2) THEN PRI
NT "*ERROR*   re-enter last line":GOTO 3
0
50 b$=LEFT$(a$,2)
60 POKE n,VAL("&"+b$)
70 n=n+1
80 a$=RIGHT$(a$,LEN(a$)-2)
90 IF a$="" GOTO 30 ELSE 50
```

Let's look at what the hexloader actually does. Line 1Ø simply sets up the memory reserve (as discussed in an earlier chapter). Line 3Ø accepts a string of hex digits and checks that an even number of characters have been given, i.e. if the string was nine characters long, then there would obviously be an error in the data entry.

Line 5Ø takes the first two characters of the string and POKEs them into the Amstrad's memory, starting at address 4ØØØØ. Line 8Ø chops off those two POKEd characters from the string, and if there are

digits left, goes back to line 5Ø where the whole process is repeated again. If there are no more digits, the program goes back and asks for another input. When you have finished your data entry, press ESC twice to break the BASIC listing.

The Program

Now let's have our first magnificent, all-powerful machine code program. We will offer it in both its mnemonic form and its hexadecimal form... are you ready for this...

RET **C9**

Not much to look at, is it? We have encountered this instruction before. It is the command for RETURNing from a subroutine and, if there are no subroutines, returning from BASIC. Therefore it is admirably suited for our purpose. If it works we will be in BASIC without any crashes and if it doesn't work then we have only lost a minute's typing. Make sure you save the Hexloader to tape before running it.

Run the hexloader, enter the hex pair and break the BASIC program. The routine has now been entered into your Amstrad's memory. To start it, you need to use the BASIC command, CALL. This must be followed by the memory location or address at which the routine starts. In this case, as with most of the routines in this book, it is 4ØØØØ. CALL 4ØØØØ will execute the program and return you to BASIC without any problem. If you do have a crash, switch your machine off, load in the hexloader and check the program listing carefully. After that, run it and make sure that you enter C9, the hex for RET.

CALL is a powerful command on the Amstrad. You will find this is so when you read the chapter a little later on passing parameters to and from machine code programs.

Variables

BASIC programs could not survive without using variables. Machine code has its own variables, known as **registers.** Registers are very limited in comparison with the freedom of using BASIC variables.

There are a number of specialist registers but for the moment we will only consider the general purpose ones. There are six of them: B, C, D, E, H and L. Each register is rather like a single memory location. It can only hold a number between zero and 255. To overcome this, we pair the registers together getting BC, DE and HL. They are capable of dealing with 16-bit numbers in the range of zero to 65535.

We have to use these registers with the various machine code instructions. The most common of these is LD, which is short for LOAD. This is equivalent to a LET in BASIC. Therefore, LET B=1Ø or B=1Ø has the machine code equivalent of LD B,1Ø (load B with 1Ø). This form of loading is known as **Immediate Addressing.**

You would find it most difficult to write a machine code routine without the LD instruction. It comes in a number of forms. As we have already seen, you can load a register with a numeric constant value. You can also LD a register with another register's value, as in LD B,C. Remember that it is always the first register that is being loaded with the value of the second register.

Accumulator

Before we go any further with the various LD's available, we need to look at the more specialised registers that exist. 'A' is known as the accumulator register. We will look at its features in chapter six dealing with simple arithmetic.

There are a number of LDs specifically for the A register. It is the only 8-bit register capable of being loaded directly from a memory location. It can be used in the same way as any of the general purpose registers in the form LD A,C or LD A,19Ø.

Let's look at these special LDs. LD A,(nn) would load A with the contents of the memory address nn. The brackets signify, 'with the contents of'. It is possible to reverse the process and load a specified memory location with the value of the A register. LD (nn),A would do this, again where nn is a memory location.

To demonstrate this a little further, let's look at an example routine. Follow the same procedure for this routine as for the earlier one, using the hex loader.

Assembler	**Hexadecimal**
LD A,(41ØØØ)	**3A 28 AØ**
LD (41ØØ1),A	**32 29 AØ**
RET	**C9**

Call 4ØØØØ will make this routine take the value of the address 41ØØØ and load it into 41ØØ1, thus performing a simple machine code copy. To check to see if this works, type

```
PRINT PEEK(41ØØØ);"   ";PEEK(41ØØ1)
```

and the numbers should be the same.

Notice how easy it would have been to have fallen into a trap at this point. Many of the routines in this book start at address 4ØØØØ. It would have been very easy to have used 4ØØØØ in this one, instead of 41ØØØ. However, by doing that, you would be placing a new value into part of the actual routine, with 4ØØØ1 holding the same instruction as address 4ØØØØ, LD A.

This should be avoided at all costs in machine code routines. This one isn't important but allowing the routine to tamper with itself when that routine is important or complex is obviously unwise.

Let's look at another couple of registers and how they are used with LD instructions. IX and IY are called **index registers.** They are 16-bit affairs and are particularly used for passing parameters from BASIC to machine code and vice versa, as discussed in the next chapter. LD IX,(addr) loads the IX register with the contents of an address.

F is the flags register, sometimes called the **status register.** From time to time this register cohabitates with A in the hope that no-one will notice it.

The **bits** of F each have a specific purpose.

They are:

Bit 7: The sign flag, or S.
Bit 6: The zero flag, or Z.
Bit 5: Not used.
Bit 4: The half carry flag, or H.
Bit 3: Not used.
Bit 2: The parity/overflow flag, or PV, or just P.
Bit 1: The subtract flag, or negate flag.
Bit 0: The carry flag, or C. (Not to be confused with register C).

The **half-carry** flag is set by arithmetic instructions if there is a carry from bit three into bit four or, in the case of register pairs, from bit eleven into bit twelve.

The **subtract** flag is set by any instruction which involves a subtraction, or reset by any instruction which involves an addition.

There are no instructions to directly alter the value of F. However, F can be made to hold a value of, say, xx using LD C,xx/PUSH BC/POP AF. Similarly, the state of the flags H and N may be read using PUSH AF/POP BC and then examining the bits of C.

Don't worry if this seems largely incomprehensible at the moment. It will become clearer in due course (we promise!), and when it does you can come back and reread this section with full understanding.

Chapter Five
Passing Parameters

We already know that the BASIC CALL command is used to access a machine code routine. CALL can be followed by more than just the address of the routine to be CALLed. This section deals with using CALL to pass parameters not only from BASIC to the machine code, but also the other way.

There are a number of types of parameters that can be passed to and from machine code. Let's look at the three most common types.

First, there is the straight whole (integer) number which can be either an actual number or an integer variable. The value will be in the range of a 16-bit two's complement number, so 31Ø92, 9, 22Ø and F% (where F% is equal to 1ØØØ) are all permitted.

When the machine code routine is called, the A register immediately holds the number of parameters following the CALL. This may or may not be of use to you. Each parameter is stored using the IX index register (discussed partly in the previous chapter and partly in the dictionary of machine code following shortly). The two byte parameter value is stored with the low byte first and the high byte second.

The register IX holds the address of the low byte of the **last** parameter. The high byte of the last parameter and the bytes of the other parameters, if there are any, are stored from address IX+1 onwards. Therefore if a CALL 4ØØØØ, 258 was entered, (IX+Ø) would hold the low byte which is 2 and (IX+1) would hold the high

byte which is 1. (If you need your memory refreshing: 16-bit numbers are calculated by 256*high byte+low byte, in this case 256*1+2=258.)

Let's say that we wanted to enter a single parameter call at 3ØØØØ with the parameter being loaded into a two-byte normal register such as HL, and the parameter was a variable, B%. Your machine code routine would start with LD L, (IX+Ø) followed by LD H, (IX+1). If it was the third from last parameter of a CALL, for example, then it would become LD L, (IX+4) followed by LD H, (IX+5). In other words, the low byte always comes first and for every parameter, there are two bytes above the IX register.

The second type of parameter is still an integer variable but, unlike the previous type, it can be passed back from machine code to BASIC as well as travelling the other way. To achieve this in the CALL, an '@' sign is added before the variable name. (Naturally, this must be a variable rather than a number.) CALL 4ØØØØ, @B% would make (IX+Ø) and (IX+1) hold the address of the value of B%.

Therefore, if you want to alter the value of B% then you first have to find the address where it is stored by examining (IX+Ø) and (IX+1). Then you must alter the value of the two memory locations holding the value of B% before returning from machine code to BASIC.

For example, if you wanted to alter the value of B% to 258 then you would load the memory location given by (IX+Ø)+256*(IX+1) with 2 (the low byte) and the memory location given by (IX+Ø)+256* (IX+1)+1 with 1 (the high byte).

Low Byte Change

Let's take a quick look at another small machine code program. This accepts your CALL with just one parameter and changes

whatever the parameter's low byte value was, to 6. If you want to use the hex loader program and load in the equivalent hex code to check the routine out please do, but it is by no means compulsory.

Assembler	Hexadecimal
LD L,(IX+Ø)	DD 6E ØØ
LD H,(IX+1)	DD 66 Ø1
LD (HL),6	36 Ø6
RET	C9

The first line takes the low byte of the parameter's address and puts it into register L. The second line takes the high byte of the parameter's address and puts it into the register H. With the address of the parameter now given by the HL register pair (by altering the memory pointed to by that register pair) we can alter the value of the variable at will. In the third line, we have changed it to 6. (In a moment I will show you a couple of other things to do with it.) In the fourth line, we have returned the computer to BASIC.

To use the routine type: B%=Ø:CALL 4ØØØØ, @B%. Printing B% after the routine has returned will show B% is valued at 6. Experimentation being the name of the game with machine code, alter the second hex pair on the third line (Ø6 at present) to another value and run the routine again. B% will now be worth that new value. By looking up the instruction op codes at the back of the book, alter the LD (HL),n instruction to LD (HL),A. This will now return the number of parameters available.

Stringing Along

The third and final types of parameter that we are going to deal with at this point are strings. These work quite differently from numeric information. For a start you can only use string variables; no CALL 4ØØØØ, "AMSTRAD" here. As well, a string variable must be

preceded by an '@' character.

The IX register holds an address concerning the parameter, as in the previous numeric cases, but this time it isn't the address of where the parameter resides in memory but the address of another memory location - the start of the **string descriptor block.** The string descriptor block holds information describing the string, in terms of the string's start address and its length.

The first byte of the string descriptor block is the length of the string and is between Ø and 255. The second two bytes form the address of where the string starts in memory, and is in the usual form of low byte first followed by high byte. Thus, this address is the address of the first character in the string. You should always define a string variable beforehand, even if it is just as A$="".

A String Routine

This tiny routine takes your string and tells you its length. To do this we need an additional parameter, an integer variable to hold the value to be returned.

Assembler	Hexadecimal
LD L, (IX+Ø)	DD 6E ØØ
LD H, (IX+1)	DD 66 Ø1
LD A, (HL)	7E
LD C, (IX+2)	DD 4E Ø2
LD B, (IX+3)	DD 46 Ø3
LD (BC), A	Ø2
RET	C9

Now execute a CALL 4ØØØØ,@V%,@A$ with both A$ and V% already defined, to return a value in V% which equals the length of the string A$.

Chapter Six
Simple Arithmetic

We will now perform some simple arithmetic in machine code. To do this, we need some new instructions.

The simplest arithmetic operations in machine code are INC (increment) and DEC (decrement). INC adds one to the specified register which can be an 8-bit one, a 16-bit register pair or an index register. In addition, this instruction can alter the value of a memory location as in INC (HL) - this is known as **indirect addressing.**

INC is a single byte command. INC A would add 1 to the value of the A register, while INC IX would do the same to the index register, IX. If you are incrementing an 8-bit register and the register's value reaches 255, then it will return to zero. If a 16-bit register pair is incremented when its value reaches 65535 then it also returns to zero.

DEC works in much the same way as INC and all the formations of instructions that are valid for INC are also valid for DEC. These are:

INC or DEC r	**(where r is an 8-bit register)**
INC or DEC rr	**(where rr is a 16-bit register pair)**
INC or DEC IX	
INC or DEC IY	
INC or DEC (HL)	
INC or DEC (IX+d)	
INC or DEC (IY+d)	

If you decrement an 8-bit register when it has a value of zero it will cycle around to 255 - the opposite procedure to INC.

Whereas the 8-bit INC and DEC instructions affect most of the flags, the 16-bit registers (when used with INC or DEC) **do not** affect any of the flags. For the 8-bit instructions the **sign flag** is set (made equal to 1) if bit 7 of the result is set to 1. The **zero flag** is set if the result is zero.

The **half carry** is set if there is a carry or borrow from bit 4 of the result. The **overflow flag** is set if bit 7 of the result is altered. The **subtract flag** is set if the last operation performed was a subtraction instruction, such as DEC. Note that the carry flag is left unaltered.

Adding and Subtracting

ADD and SUB are examples of easy-to-understand mnemonics. It is pretty obvious that they stand for add and subtract.

ADD and SUB only work with one 8-bit register, the A register and with the HL register pair and the two index registers, IX and IY. Apart from their functions, which differ, the syntax and uses are almost the same.

It is a useful idea to show ADDs and SUBs emulating BASIC commands. This gives you a better, clearer idea of how you can use them within routines:

LET A=A+9	**ADD A,9**
LET A=A+C	**ADD A,C**
LET A=A-C	**SUB A,C**
LET A=A-178	**SUB A,178**

Only the A register can work directly with these commands. ADD B,9 or ADD C,D are not allowed. Therefore, if you wish to perform

arithmetic with the other registers, you have to use a procedure such as this one, which will subtract 8 from the register, E:

```
LD A,E      (Load A with E)
SUB 8       (Subtract 8 from A)
LD E,A      (Load E with A, the new value of E)
```

Whenever an addition or subtraction is made and 8-bit registers are used, you need to make sure that the result of the arithmetical operation can be stored within an 8-bit number.

For those operations that cannot, 16-bit addition or subtraction may work. Most 16-bit arithmetic is performed on the HL register pair. This can have other register pairs added to it and, similarly, subtracted from it as shown:

```
ADD HL,BC
ADD HL,DE
ADD HL,HL
ADD HL,SP
```

Note that the second register pair specified is not changed by these operations. The other 16-bit arithmetic operations work on the index registers as shown:

```
ADD IX,BC
ADD IX,DE
ADD IX,IX
ADD IX,SP
```

IY is handled the same way. There is no instruction to add a 16-bit number to HL directly. One way of doing this would be to load DE with the desired number and then use ADD HL,DE but this corrupts the DE register pair. The carry flag is also affected by these operations. So, if the result obtained is larger than 65535 the carry

flag is set, otherwise it is reset.

Carry

There are two more instructions dealing with addition and subtraction. These work in exactly the same way as ADD and SUB except that they make use of the **carry flag.** These are ADC (add with carry) and SBC (sub with carry). In the case of 8-bit arithmetic they only work on the A register as before. With 16-bit arithmetic they work on HL only, and not on IX or IY. ADC and SBC simply include the carry in the calculation, so the following operations are identical.

LET A=A+5+carry	**ADC A,5**
LET A=(A-12)-carry	**SBC A,12**

These instructions are very powerful as they allow you to perform arithmetic operations on, say, two 32-bit numbers. Sometimes you will need to make sure that the carry flag is reset before using them. One way of doing this is by using the instruction AND A which does not alter the value held in A but it does reset the carry flag as a carry can never occur.

No Division

You may well have noticed that there are no division or multiplication instructions. If your program requires many complex mathematical formulae and calculations, then it may be best to write those parts in BASIC. One disadvangage of machine code is its poor ability to handle complex mathematical functions.

However, it is possible to emulate multiplication very easily, using the ADD instruction. The A register will hold the original value at

first and then later will hold the answer after the original value has been multiplied.

Multiplication by 2 is very simple: ADD A,A

In fact, any number that comes within the range of an 8-bit number and is a member of the family, such as 2x2=4x2=8x2=16x2=32, can be used as a multiplier very easily.

```
ADD A,A
ADD A,A = multiplication by 4 and
```

```
ADD A,A
ADD A,A
ADD A,A = multiplication by 8
```

You can continue this so that four ADDs are needed to multiply by 16, five for 32 and so on.

Other multipliers are not quite so easy to get but need the use of the LD command and in some cases the SUB command. Even so, they make up for only a few bytes of instructions and could be well worth using in your machine code routines of the future. Here are a couple of examples:

```
LD B,A
ADD A,A      =      x 3
ADD A,B
```

```
LD B,A
ADD A,A
ADD A,A      =      x 7
ADD A,A
SUB A,B
```

Note the last instruction in the times 7 routine. If you look at the Op Codes table you will see that it is not actually there. What happens is that SUB B is exactly the same as SUB A,B. You can leave the A out of the notation as it is only the A register that can legally be in that position. This applies to all the SUB operations that seem to only have one register.

Chapter Seven
Stacking and Jumping

The Stack

There is an area of RAM that is set aside for storing various pieces of information to help the machine know what it's doing. It works as follows:

The word "stack" is something that the computer people have got straight out of a dictionary. It means exactly what it says. Imagine a stack of cardboard boxes. Each box is really a memory location, so each has an address - but if you want to know what's inside any particular cardboard box, the only one you can look at easily is the top one. If you tried to pull one of the boxes from somewhere in the middle then all the boxes above it would fall down. Conversely, to add a new box to the stack the only place you can put it easily is at the top.

The memory locations in the stack are just like that. You can put things on top of it, but **only** at the top, and you can take things **from the top.**

There are two special words which go with the stack - one which means "stacking a new number onto the top" and another which means "removing a number from the top"; the first word is PUSH, and the second word is POP. If you PUSH the number five onto the stack, and then you PUSH the number 9ABC, and then you PUSH, say, 8ØØØh, then the first number you can POP is 8ØØØh, since this number is now at the top (because it was PUSHed there last); the

second number you can POP is 9ABC, and the third is five.

The stack is stored very high amongst the Amstrad's addresses, so that there is less chance of the BASIC program "colliding" with the stack as either one or the other is built up. The stack is actually very peculiar, because it's **upside down**! It turns out to be more efficient this way. So remember, the stack, or **machine stack** as it's sometimes called, is like a stack of cardboard boxes piled upon a shop floor, except that in a daring feat of defiance of Newton's laws this stack instead decides to reside on the ceiling and build up downwards! The top – the only part you can actually get at – is lower than the bottom!

The stack is so important to the computer that a special register is set aside just to store the position of the **top** of the stack (the part with the lowest address – the part we can get to). That register is called SP, which stands for Stack Pointer. It is actually a register pair because it stores two separate bytes, but unlike the other register pairs (BC, DE and HL) we **cannot** separate the two halves – they really are glued together quite solidly.

Here's how the instructions PUSH and POP work. I'll do this in hex because I think it's easier that way. Suppose HL contained a value of ABCD. This means that H contained AB and L contained CD. Now, the instruction PUSH HL would store the number ABCD at the top of the stack. It would do so by first of all stacking the **high** part (AB), and then stacking the **low** part (CD). It would alter the value of SP accordingly since two more bytes have been added to the stack, and the position of the top will therefore have moved (down) by two addresses.

It is unfortunately not possible to PUSH single registers onto the stack. You may only PUSH register **pairs,** so BC may be PUSHed, but B on its own may not. It is worth noting that PUSH BC will not in any way alter the value of BC, but will simply copy it without changing it. This, of course, goes for all PUSH instructions.

PUSH can be thought of in BASIC as three separate statements:

```
PUSH HL     SP=SP-2
            POKE SP+1,H
            POKE SP,L
```

POP, of course, works the other way round. POP HL will first of all remove L from the stack, and will then remove H. SP will be changed, since the top of the stack will have moved.

```
POP HL      L=PEEK SP
            H=PEEK (SP+1)
            SP=SP+2
```

Verifying, using the BASIC equivalents given, that PUSH HL followed by POP DE is the same thing as LD D,H followed by LD E,L.

PUSH

Here are the codes for the instruction PUSH. One of them will require a small degree of explanation.

```
F5          PUSH AF
C5          PUSH BC
D5          PUSH DE
E5          PUSH HL
```

The register pair AF, which normally cannot be used as such, is made up of the smaller single registers A and F, in the same way that BC is composed of B and C. A is a register which we've been using, but F is something completely different. The F stands for Flags. To understand the workings of F you have to look at it not in hex, but in **binary.** F could, for instance, have a value of 41h, which in binary is

Ø1ØØØØØ1. Each of the digits is either zero or one, and each of the different digits has a different meaning for they are all different flags. One of these flags, the carry flag, is actually stored as the rightmost binary digit of F.

POP

The codes for the POP instructions are very similar to the codes used for PUSH. They are:

F1	**POP AF**
C1	**POP BC**
D1	**POP DE**
E1	**POP HL**

One of the major uses of PUSH AF and POP AF is simply to PUSH and POP the value of A. The fact that F has been stacked alongside it can be conveniently forgotten about. PUSH AF will certainly stack the value of A until it's needed again, at which point it may be recovered by the use of POP AF. This could be useful if you have to use the A register to perform calculations of some kind that could not be performed in another register, but where the value of A will still be needed later on in the program.

For example, to add twenty-five to the value of B without altering the value of A (or any other register):

F5	**PUSH AF**
78	**LD A,B**
C619	**ADD A,19h (=25d)**
47	**LD B,A**
F1	**POP AF**

Why will only B and no other register be altered? (Not even the carry

flag!) See if you can work out precisely what the above routine is doing, before you read on.

Altering SP

We can actually use SP in much the same way that we use BC and DE. We can add and subtract it, and we can load it. The hex codes are:

F9	**LD SP,HL**
31????	**LD SP,nn**
ED7B????	**LD SP,(nn)**
ED73????	**LD (nn),SP**
39	**ADD HL, SP**
ED7A	**ADC HL,SP**
ED72	**SBC HL,SP**
33	**INC SP**
3B	**DEC SP**

This is very powerful, and very useful. Suppose you wanted to exchange the values of D and E without altering anything else. The following routine will do just that.

D5	**PUSH DE**
D5	**PUSH DE**
33	**INC SP**
D1	**POP DE**
33	**INC SP**

The final INC SP was necessary in order to restore the stack pointer to its original value. If this is not done you may get a pretty nasty crash.

SP is not the only specialised register in use. There is another two

byte register called PC, standing for Program Counter. Its job is to remember whereabouts we are in the program. Every time the Amstrad has to execute an instruction it will take a look at what PC says. If it says A00Ø4 then it will execute the instruction at location A00Ø4. It will then increment the value of PC by the number of bytes in that instruction, so that **next** time round it will be looking at the next instruction in sequence. For example, if A00Ø4 contained the instruction LD A,B, then this would be carried out and PC would be incremented to A00Ø5. If the instruction at A00Ø5 was LD B,2 then once this was carried out, PC would be increased by **two,** since LD B,2 is a **two byte** instruction. PC would then be reading A00Ø7 where the next instruction begins.

If you alter the value of PC then the effect is like a BASIC GO TO. The only difference is that machine code does not use line numbers, so you have to GO TO the right **address** rather than the right line number. The machine language instruction to do this is called JP which, of course, is short for JumP. JP A000 means GO TO address A000 and continue executing the machine code from there. Of course, all this instruction really does is to load the number A000 into PC (but without incrementing it at the end of the instruction) so that it thinks A000 is the next address in the program. It is far more useful for us human beings to think of it as a kind of GO TO though, because that's what we're used to.

Be Careful

Be careful with JP though. If you create an infinite loop in machine code then **tough**! You're stuck with it – you can never break out unless you actually reset the machine or switch it off. An example of an infinite loop would be:

77	**A000**	**LD (HL),A**
23	**A001**	**INC HL**
C300F0	**A002**	**JP A000**

I've written the actual addresses in the middle column. Usually this isn't done and important lines are marked with **labels,** or words which tell us which lines do what. These labels do not appear in the hex, and we only in fact write them out for our own convenience. If, for instance, we decided to call the first line START then our pretty bad program could be written as:

```
77          START       LD (HL),A
23                      INC HL
C3ØØFØ                  JP START
```

There is another instruction (similar to JP) called JR, or Jump Relative. It means jump forward a given number of bytes. In many ways it is better than JP because it is only two bytes long instead of three, and because a whole routine may be relocated without changing JP destinations all over the place. JR Ø has no effect whatsoever, and the next instruction will be executed in sequence; however, JR 1 will cause the next instruction (assuming it to be a single byte instruction) to be skipped. To skip over a two byte instruction, or two single byte instructions, you will need to use JR 2.

It is also possible to jump **backwards** using JR, since there is a convention that any hex number between 8Ø and FF will be taken to mean a negative number (actually 256d less than the number it would normally represent). Note that the number minus five, for example, is represented by FB, and so therefore it is possible to use the instruction JR – 5 – but note that because of this convention we are unable to say JR 129d, for instance, because 129d in hex is 81, which would be taken to mean – 127d and would be a jump backwards. The range we are limited to is, therefore, – 128d to 127d.

Doing Nothing

JR Ø, as has been said, does absolutely nothing. It will continue with the next instruction. It is important to remember that relative jumps

are counted from the **next** instruction. JR Ø means execute the **next plus zero** instruction. JR 1 means execute the **next plus one** instruction. Consequently, if we were to say JR – 2 then you must count backwards for two bytes starting at zero with the **next** instruction. You will find that two bytes leads you to precisely the instruction we have just executed – the instruction JR – 2. JR – 2 is therefore an infinite loop all by itself, and is thus not really a recommended instruction to use in a program.

The (rather silly) infinite loop program above can now be rewritten in one byte less using JR instead of JP.

```
77          START       LD (HL),A
23                      INC HL
18FC                    JR START
```

Notice how I wrote "JR START" instead of "JR – 4". This makes the program much easier to follow since all we have to do is look for the label START in order to know where the JR is taking us.

JR and JP are more or less useless on their own without conditions attached, in the same way that BASIC GO TO would be useless if the instruction IF/THEN GO TO were not around. We need some kind of **conditional** jump, so that we can say IF some condition is true THEN jump to a new address. Without this facility JP and JR can only really be used to create infinite loops. Although machine code does not have quite the same degree of flexibility as BASIC, it does allow us to check for four conditions on JR, or eight conditions on JP. These are (for JR):

18ee	**JR e**	**Jump Relative by e bytes.**
20ee	**JR NZ,e**	**IF the last result calculated was non zero THEN Jump Relative by e bytes.**

28ee	**JR Z,e**	**IF the last result calculated was zero THEN Jump Relative by e bytes.**
30ee	**JR NC,e**	**IF CARRY = Ø THEN Jump Relative by e bytes.**
38ee	**JR C,e**	**IF CARRY = 1 THEN Jump Relative by e bytes.**

And for JP:

C3qqpp	**JP pq**	**Jump to address pq.**
C2qqpp	**JP NZ,pq**	**IF the last result calculated was non zero THEN jump to address pq.**
CAqqpp	**JP Z,pq**	**IF the last result calculated was zero THEN jump to address pq.**
D2qqpp	**JP NC,pq**	**IF CARRY = Ø THEN jump to address pq.**
DAqqpp	**JP C,pq**	**IF CARRY = 1 THEN jump to address pq.**
E2qqpp	**JP PO,pq**	**See below.**
EAqqpp	**JP PE,pq**	**See below.**
F2qqpp	**JP P,pq**	**IF the last result calculated was positive (Plus) THEN jump to address pq.**
FAqqpp	**JP M,pq**	**IF the last result calculated was negative (Minus) THEN jump to address pq.**

Now although this is a far cry from IF A$="HELLO" THEN PRINT "GOODBYE" as you're used to, you'll soon see that even this horrendous task may be carried out in machine code. First though, I'll explain about two of the instructions in the list above – JP PO and JP PE. Straightforwardly, JP PO means IF PV=Ø THEN jump to address pq, and JP PE means IF PV=1 THEN jump to address pq. But what exactly is PV?

Answer: PV is another flag – just like CARRY. It can only ever store one of two values – one or zero. The P stands for Parity and the V stands for oVerflow, because computer boffins, like all good mathematicians, can't spell properly. This usage is quite easy to explain:

JP PO can also be thought of as JP NV (Jump if No-oVerflow).
JP PE can also be thought of as JP V (Jump if oVerflow).

Technically you shouldn't write JP NV or JP V because it's not standard convention, but it is certainly an aid to memory. I don't really think it matters if you're conventional or not as long as you know what it means (NV=PO and V=PE).

Now, to **overflows.** If we regard numbers from 8Ø to FF as negative and numbers from ØØ to 7F as positive then strange things can happen in arithmetic if we try to cross that boundary. For instance, 41h (positive) plus 41h (positive) equals 82h (negative?). This sort of blunder is called an overflow, so JP V will jump if such an overflow has occured and JP NV will jump if such an overflow has not occurred. In simple terms, an overflow has occurred if the result of any arithmetic operation acting on numbers in positive/negative convention (also called "two's complement" convention) gives an answer with "the wrong sign".

To repeat once more: JP NV is a non-standard way of writing JP PO, and JP V is a way or writing JP PE. So NV=PO and V=PE. You should invent some way of remembering this.

All of these various tests, if combined with other instructions properly, can really check for any situation conceivable. In fact, there's only one other kind of instruction you need in order to make JP and JR as powerful as IF/THEN/GO TO – that instruction is called CP, or ComPare.

CP will compare the register A with any other register or with any numeric constant. For instance, we could have CP B (Compare A with B) or CP L (Compare A with L) or CP 3E (Compare A with the number 3E). What this instruction actually does can be thought of in BASIC as, for CP B, DUMMY = A – B.

In other words, a subtraction is performed, but the **result** of that subtraction is "forgotten", and the value of A never changes. The flags, however, **will** change and tell us all about the result. If A contains Ø5 and B contains Ø6 then after a CP B instruction JP NZ will work (since Ø5 – Ø6 is non zero), JP C will work (since Ø5 is less than Ø6), JP NV will work (there is no overflow since the result of Ø5 – Ø6 = FF which is negative and which is **supposed** to be negative) and JP M will work (since FF is negative). Although the subtraction is actually carried out it should be emphasised that the result of the subtraction is discarded, and the value of A does not change.

You can do some useful tricks with CP:

If A=B THEN GO TO...	**CP B / JR Z ...**
IF A<B THEN GO TO...	**CP B / JR C ...**
(This will only work if all numbers are assumed positive)	
IF A<B THEN GO TO...	**CP B / JP M...**

CALLing

Even in machine code we can have **subroutines.** The machine code equivalent of GO SUB is called CALL. We write CALL pq to mean GO SUB the subroutine starting at address pq. The equivalent

RETURN instruction is RET. I hope this looks familiar. RET has a dual purpose – at the end of a subroutine it means "return from that subroutine"; if there is no subroutine to return from, it means "return to BASIC". CALL and RET can both have conditions imposed on them as follows:

CDqqpp	**CALL pq**	**C9**	**RET**
C4qqpp	**CALL NZ,pq**	**CØ**	**RET NZ**
CCqqpp	**CALL Z,pq**	**C8**	**RET Z**
D4qqpp	**CALL NC,pq**	**DØ**	**RET NC**
DCqqpp	**CALL C,pq**	**D8**	**RET C**
E4qqpp	**CALL PO,pq**	**EØ**	**RET PØ**
	"CALL NV,pq"		**"RET NV"**
ECqqpp	**CALL PE,pq**	**E8**	**RET PE**
	"CALL V,pq"		**"RET V"**
F4qqpp	**CALL P,pq**	**FØ**	**RET P**
FCqqpp	**CALL M,pq**	**F8**	**RET M**

As you'd probably guessed from the above, instructions like RET Z, etc., can also be used to conditionally return to BASIC, ie RET Z equals IF zero THEN RETURN to BASIC.

In BASIC there is no stack, so we never need to worry about it. In machine code, however, there is and so we **do** need to worry about it. There are two instructions other than PUSH and POP which alter the stack! These are CALL and RET!

CALL pq is equivalent to PUSH "the-address-of-the-next-instruction-to-be-executed", followed by JP pq.

RET is equivalent to POP DUMMY followed by "jump-to-address-DUMMY".

You should be able to see how this procedure causes CALL and RET to act as they do. Whilst this is certainly efficient, it does have some drawbacks which we must watch for.

The value of SP must not be altered during the course of a subroutine, since both CALL and RET rely on the stack. You can PUSH as many items as you like onto the stack during a subroutine, just so long as you POP an equal number before attempting to return. A clever trick you may like to know is how to actually **alter** the return address from a subroutine. Let's say you want to change the return address to BØØØ. Watch this:

E1	**POP HL**
21ØØEØ	**LD HL,BØØØ**
E5	**PUSH HL**

The first instruction deletes the original return address from the stack. The second and third instructions replace it by a new alternative return address. When, at any later stage, a RET instruction is reached, control will "return" to address BØØØ. Another useful trick to know is how to make absolutely sure that your subroutines will always exit with the stack "balanced". One way of doing this is to store the value of the stack pointer somewhere and then retrieve it at the end.

Chapter Eight
A Dictionary of Machine Code Terms

This chapter should form a handy refresher after the tutorials of the last few chapters. This dictionary contains a short description of all the Z8Ø Op Codes, the registers and a type of command called a directive which is found in assembler listings. These summaries give you the chance to check up on particular commands or features that you may not have fully understood as well as providing a useful reference source for the future when you come across an obscure, infrequently-used command or register.

The dictionary also includes details of the Z8Ø Op Codes that we haven't yet covered. Reading about them here will give you a good understanding of how they work and are used.

Registers

In addition to the registers A,B,C,D,E,H and L, and the sign flags, PV and carry, there are a couple of other registers and flags, although not all of them are useful. Let's cover the registers first.

IX is a register pair; however, it may not be split into its component bytes like HL can. Any instruction in machine code which involves HL (not in brackets) may be written with IX instead of HL. (There are three exceptions to this rule: EX DE, HL, ADC HL, and SBC

HL,.) The hex code of such an instruction is DD followed by the hex code of the corresponding instruction using HL. Any instruction in machine code which involves (HL) (with brackets) may be rewritten with (IX + dd) instead of (HL) – the dd represents any byte. For instance, since there is an instruction LD (HL),Ø3 then there is also an instruction LD (IX + 2A),Ø3. The displacement byte can be very useful. (There is an exception to this rule: JP (HL) may only be rewritten as JP (IX) – not JP (IX + dd).) The hex code of such an instruction is DD followed by the hex code of the corresponding instruction, but with the displacement byte inserted into the third byte of the hex code. For example, the hex code for LD (HL),Ø3 is 36Ø3; therefore, the hex code of LD (IX + 2A),Ø3 is DD362AØ3.

IY is also a register pair. It is used in exactly the same way as IX except that the hex codes of IY instructions use the byte FD rather than DD. Despite the fact that both of these register names begin with I they do **not** have the high part in common, and are both independent of each other.

SP is the stack pointer. It's a register pair like BC or DE, however, its two component bytes may not be separated. SP **always** points to the topmost item on the machine stack. If you change the value of SP then you automatically create a new stack at the given address. Addresses immediately below the current value of SP are likely to be overwritten without prior consent by something called the Z8Ø's interrupt handling routine (see DI).

A is a register you should all know about by now. It's called the **accumulator** from time to time, since it may be used in one or two ways in which the other registers may not (eg ADD A,Ø6).

B,C,D,E,H and L. These are registers which you should by now be totally familiar with.

The Alternate Register Set, featuring A′, B′, C′, D′, E′, F′, H′ and L′ is of little use except for copying the contents of your normal

registers to for safekeeping. However, Locomotive BASIC uses these registers and this plus Amstrad's warning not to use the alternate register set makes it advisable to leave them well alone.

F register holds all of the flags as discussed earlier.

All the Instructions

By now we've seen a fair number of Z8Ø instructions, so you'll be wanting to expand your vocabulary of these. Here now is a detailed list of every single instruction available to you. I shall cover them in alphabetical order so that you may use this chapter as a kind of dictionary of machine code instructions. For precisely that reason I shall also go over the ones we've already seen. You should, however, re-read them as this will provide a useful memory aid.

ADC Starting with ADC. It comes in two forms: ADC A,r and ADC HL,s. Here we are using r to stand for either A,B,C,D,E,H,L, a numerical constant, or the contents of an address (HL), (IX + d) or (IY + d). ADC A,r is a single byte instruction. It calculates the sum A plus r plus CARRY. The result is stored in A. ADC HL,s is a two byte instruction which evaluates HL plus s plus CARRY, and stores the result in HL. s here stands for either BC, DE, HL,IX or IY. Can you see why (ignoring flags) ADC A,A does precisely the same job as RLA?

ADD Very similar to ADC except that the carry flag is not used in the initial evaluation. It is, however, still altered by the final result. There are two important differences between ADC and ADD though. Firstly, the set of instructions ADD HL,s (where s means as it did in the description of ADC) are one byte instructions rather than two and secondly, it is permissible to use two further sets of instructions ADD IX,s and ADD IY,s.

AND Only one form here – AND r. The value of the A register is altered one bit at a time. If such a bit is zero it will remain unaltered, otherwise it will take on the value of the corresponding bit of r. Thus, AND ØØ will always result in zero and AND FF will leave A unchanged. AND alters each of the flags. Specifically, the carry flag will always be reset to zero.

BIT The form of this instruction is BIT n,r where n is a number between zero and seven. The instruction alters the zero flag (only) according to the current value of the bit in question. If the bit is zero then the zero flag will be set, otherwise the zero flag will be reset. You can exploit this using instructions like JR Z (which will jump if the bit was zero) or RET NZ (which will return if the bit was not-zero). BIT does not alter the value of any of the registers, nor does it change the value of the carry flag. It is a two byte instruction. I tend to find it's not used all that often, but that when it is used it comes in very handy indeed.

CALL You've seen this one before – it's rather like GO SUB. Its exact function is as follows: PUSH the return address onto the stack, and then jump to the call address. Since the return address (now on the stack) is used by the instruction RET, it is vitally important that a subroutine should not alter the stack. You may only PUSH things onto the stack in a subroutine if you POP them off again before you attempt to return. CALL may also be used with conditions – for example, CALL Z,pq (pq is an absolute address) which means IF the zero flag is set then CALL pq, otherwise continue with the next instruction.

CCF Complement Carry Flag. If the carry flag was zero then set it to one. If it was one then reset it to zero.

CP In the form CP r, it will calculate the result of subtracting r from A, but will not store the result anywhere. The previous value of A (and, of course, r) remains unaltered. It will, on the other hand, alter

all of the flags, so conditional instructions like JP Z or JP C will still work. CP r followed by JR Z will jump if A equals r (since A minus r is zero) and so on.

CPD Imagine this as CP (HL) followed by DEC HL followed by DEC BC. The PV flag is reset if BC decrements to zero and set otherwise. The zero flag is set if CP part of the instruction found that A was equal to (HL), otherwise it is reset.

CPDR Basically this is the same as CPD except that the instruction is executed over and over again – a kind of automatic loop if you like. CPDR stands for ComPare with Decrement and Repeat. The loop will end in one of two cases: (i) either the comparison found A equal to (HL) (ie if now A = (HL + 1)) and the zero flag is set, or (ii) BC reaches zero, in which case the PV flag is reset.

CPI As CPD, except that HL is incremented instead of decremented.

CPIR As CPDR, except that HL is incremented instead of decremented.

CPL An abbreviation for ComPLement. The register A is altered bit by bit. If any particular bit starts off as zero then it is changed to one and **vice versa.** In other words, if A started off as 11Ø1Ø1Ø1 (binary) then CPL would change it to ØØ1Ø1Ø1Ø (binary). This is equivalent to subtracting A from FF. The flags are not affected by this instruction.

DAA Suppose you wanted to add sixteen to twenty-six **without** converting the numbers to hex. The following seems plausible: LD A,16 then ADD A,26. Unfortunately, because the machine works in hex the final value of A will be 3C not 42. The instruction DAA (Decimal Adjust Accumulator) will then change A from 3C to 42. How it works is rather complicated – it makes a note of what's been carried where and whether you've added or subtracted and so on;

but it does always work. For instance, the sequence LD A,42 then SUB Ø6 will again leave A with 3C, but this time round DAA will change A to 36, since forty-two minus six is thirty-six. The instruction changes every flag appropriately.

DEC This is another one of those instructions which comes in two forms. It can be DEC r (a single register) or DEC s (a register pair). DEC r is very simple to understand – the value of the register r is decremented (decreased by one, or changed from ØØ to FF), the carry flag is unaltered and the zero flag is changed as you'd expect it to be. DEC s, however, is a sneaky little instruction, for the zero flag is **not** altered! In fact, none of the flags at all are changed! Thus DEC BC/JR NZ, – 3 is either an infinite loop or has no effect! You must be very careful to remember this – a lot of our earlier programs crashed because we didn't!

DEFB Technically speaking this isn't really a machine code instruction – it's what's called a directive. The word DEFB must be followed by one or more bytes of data, each separated by a comma. Usually this data is in hex, but this isn't always necessary, eg DEFB 3A,C1,45d,11Ø1111Øb,"f", **yellow** is valid. The data is inserted into the machine code program at the point it occurs, and in the order it's listed. Data which forms part of a machine code program should never be executed, since the Z8Ø cannot distinguish between data and program.

DEFM Similar to DEFB except that the data which follows the word DEFM must be a string of characters surrounded by quotes. Commas in the text will themselves be interpreted as data, not as separators. For instance, DEFM"SOLSTICE" will cause the bytes 53 4F 4C 53 54 49 43 45 to be inserted into the program. DEFM stands for DEFine Message, as opposed to DEFB which means DEFine Byte(s).

DEFS Yet another of those directives. This one means DEFine

Space(s). The word DEFS should be followed by one numerical constant. (Just one, mind.) The given number instructs the computer to insert **that number** of zeros. So, DEFS Ø8 will insert eight bytes into the program at that point. DEFS is used mainly to define "variables" in RAM; eg FRED DEFS Ø2 (FRED is a label) and then at some later point in the program LD (FRED),HL.

DEFW One last directive (for now). DEFW stands for DEFine Word. It is used in a very similar way to DEFB except that the items of data are two bytes long, not one, so that (for instance) DEFW 4ØØØ is equivalent to DEFB ØØ,4Ø. Note how the bytes are switched around. Labels and expressions are permitted in DEFW, so DEFW 7ØØØ, FRED, ERIC + 3 is quite valid.

DI This stands for Disable Interrupts and, although this sounds pretty confusing, its use is immensely simple. Fifty times a second a little pulse is sent down one of the pins of the Z8Ø chip. There is a flag called IFF1 which stands for Interrupt Flip Flop One (no, I didn't make that up!), and the effect of DI can be thought of as RES IFF1. When the Z8Ø receives one of these pulses it checks the value of the flag IFF1. If it is set then the computer acts as if a RST 38 instruction (or CALL ØØ38) has just been reached, with the return address being the next instruction in sequence. If IFF1 is **reset** however, then no action is taken and any machine code program will run as normal. the flag IFF1 must be set **before** any attempt is made to return to BASIC.

DJNZ Yet another abbreviation. This one stands for Decrement B and Jump if Not Zero. So if B is 7, DJNZ will reduce it to 6 and then jump to a new destination. If B is zero, DJNZ will reduce it to FF and again jump to a new destination. If B is one, however, then DJNZ will change it to zero and will **not** jump to a new destination. Instead, it will simply carry on with the next instruction. The form of the instruction is DJNZ e, where e is a single byte. If B is decremented to zero then the e is ignored. If not, then the e specifies how

far to jump. The displacement calculated is as in JR.

EI Guess what? Another abbreviation. EI stands for Enable Interrupts and is the opposite of DI. This instruction is equivalent to SET IFF1. See DI for a more complete explanation.

EQU Short for EQUate. This is **not** a machine code instruction, but a directive. Each EQU directive must have a label attached to it, and the word EQU must be followed by a number (in the range ØØØØ to FFFF) or an expression such as ERIC+2. When this directive is reached no action is taken, and no bytes are inserted into the program – therefore, it doesn't really matter whereabouts in the program an EQU appears although it is conventional to place all EQU directives at the very start of a program. What it does is to assign a numerical value (the one given) to the label attached. In other words, if you have ANNIE EQU 9ØØØ and then at some later stage LD HL,(ANNIE), then the instruction will be compiled as LD HL,(9ØØØ).

EX Short for exchange. This instruction will swap the values contained by certain register pairs. There are five EX instructions – these are EX AF,AF'; EX DE, HL; EX (SP), HL; EX (SP),IX; and EX (SP),IY. They don't alter any of the flags, all they do is swap the values over – thus, EX DE,HL replaces DE by the value HL used to contain and HL by the value DE used to contain. The last three are rather interesting – the old value of HL (or IX or IY) is swapped with the topmost item on the stack, so LD BC,Ø123/PUSH BC/LD HL,4567/EX (SP),HL/POP BC will leave BC with 4567 and HL with Ø123. EX (SP),HL does not move the stack pointer and nor, of course, do EX (SP),IX or EX (SP),IY.

EXX You can imagine this as EX BC,B'C' followed by EX DE, D'E' followed by EX HL,H'L'. Basically, each of the common registers (except A) is exchanged with its corresponding alternative register.

HALT When a HALT instruction is reached, control will wait at that point in the program until the next interrupt occurs. When this happens the instruction RST 38 (CALLØØ38) will be executed and on return, control will then continue from the first instruction after HALT. Note that the flag IFF1 **must** be set if a HALT is to be executed, otherwise an interrupt will never occur. In this case, HALT will literally halt forever. There will be no way of breaking out except by pulling out the plug.

IM ***DANGER!!!*** Under no circumstances use this instruction.

IN This has two forms. The first is IN A,(n) where n is a numerical constant. This is equivalent to saying LET A=IN(256*A+n). The second form is IN r,(C) where r is a register. This is equivalent to saying LET r=IN(256*B+C). The argument of IN refers to a hardware device outside the Z8Ø chip – a different number for each different device. In the form IN A,(n) the flags are not altered; however, the flags **are** altered by IN r,(C).

INC Don't panic! At long last we're back to sensible instructions we can all understand. INC r increases the value of the register r by one, but without altering the carry flag. INC s increases the value of s by one but alters no flags at all.

IND IN with Decrement. IND can be thought of as IN(HL),(C) followed by DED HL followed by DEC B. The carry flag is unaltered, but the zero flag reflects the new result of B.

INDR As IND, except that the instruction re-executes over and over again, stopping only when B reaches zero.

INI As IND, except that HL is incremented instead of decremented.

INIR As INDR, except that HL is incremented instead of decremented.

JP If you can understand GO TO 1Ø then you can understand JP 73ØØ. The destination is an address, not a line number, but the principle is exactly the same. JP is the machine language GO TO. We also have conditional jumps, for example JP NZ,73ØØ means IF non zero THEN jump to address 73ØØ (In other words if the zero flag is not set.) There is another form of JP which also has an analogy in BASIC - variable destinations. If you understand GO TO N then you'll understand JP (HL). JP (HL) means GO TO HL. In this form you may not have conditions: for instance, JP NC,(HL) is not allowed. Only one of three register pairs may be used as variable destinations - these are HL, IX and IY. Even so these are very powerful instructions - HL can be the result of a calculation, possibly even generated at random.

JR The same as JP only slightly less powerful, but one byte shorter. Only four of the eight conditions may be used: Z, NZ, C and NC. This means it is impossible to use (for instance) JR PO. It is also impossible to say JR (HL). JR does not use an absolute address - the R stands for Relative. You write the instruction as JR e (or JR Z,e or whatever) where the e is a single byte which specifies how far to jump. JR Ø has no effect, since it jumps forward by zero bytes. JR FE is an infinite loop because control will jump back to the JR FE instruction itself. The displacement byte starts counting from the instruction immediately after the JR e instruction. If the byte is between ØØ and 7F then the jump is forward, and if the byte is between 8Ø and FF then the jump is backwards.

LD The most used instruction in the whole of machine language. All it does is to transfer data from one location to another. It has many, many forms: the simplest being LD r1,r2 - that is to transfer data from one register to another. Other forms are LD A,(BC), LD A,(DE), and LD A,(HL) - and in reverse, LD (BC),A, LD (DE),A and LD (HL),A. Remember that the brackets mean the **contents** of an address. Registers I and R may be loaded, in conjunction with A (but only A) the registers and the register pairs may all be loaded with numerical constants, the register pairs with the contents of any

address and inversely any address may be loaded from a register pair (– note that register pairs store two bytes, not one, and these are transferred to and from the address pointed to **and the address pointed to plus one**). Also allowed are LD A,(pq) and LD (pq),A (where pq represents an address) and SP may be loaded from HL, IX, IY or (pq). ((pq) may be loaded from SP but HL, IX and IY may not.) In other words – there's a lot you can do and a lot you can't. You can't say LD HL,DE, for instance (– you must use LD H,D then LD L,E or vice versa). Fortunately, since LD is used so very, very often, it is extremely easy to become familiar with its many forms.

LDD LoaD with Decrement. Effectively, LD (DE),(HL) followed by DEC HL, DEC DE and DEC BC all in one go. The carry flag and zero flag are unaltered as is the sign flag, however the PV flag is reset to zero if and only if BC decrements to zero. Thus, JP PO will jump only if BC is zero after the instruction.

LDDR As LDD, but the instruction is executed repeatedly until BC reaches zero.

LDI As LDD except that DE and HL are both incremented instead of decremented. BC is still decremented as before.

LDIR As LDI, but the instruction is executed repeatedly until BC reaches zero.

NEG NEGates the accumulator (or A register). It works by performing the subtraction ØØ minus A and all the flags are changed accordingly. Thus, S reflects the sign of the result. Z will be set if and only if A is zero. P will be set if and only if A is 8Ø. C will always be set unless A is zero. NEG is equivalent to CPL followed by INC A (ignoring flags).

NOP This wondrous little beastie (whose name incidentally is short

for No OPeration) has a very simple purpose – to waste time. It has two major uses: (i) as a delay, or (ii) to overwrite previous machine code when debugging or editing. I suppose its nearest BASIC equivalent would be a blank REM statement.

OR In the form OR r this instruction is almost the opposite of AND r. Bit by bit the value of the A register is changed. If any given bit is **one** then it remains unaltered, otherwise it takes its new value from the corresponding bit of r. If A contains ØØ then (ignoring flags) OR r is the same as LD A,r. OR FF is effectively LD A,FF. All of the flags will change as you'd expect them to and the carry flag is reset to zero.

ORG ORG is a directive which must **not** have a label attached. The word ORG must be followed by a number in the range ØØØØ to FFFF. It means all machine code from here on is to be written to the address given. Thus, ORG 7ØØØ followed by LD A,Ø1 means that the instruction LD A,Ø1 is to reside at address 7ØØØ. Unless the next thing encountered is another ORG directive, then the next instruction will be at address 7ØØ2 (since LD A,Ø1 is a two byte instruction).

OUT The OUT instruction has two forms. The first is OUT (n),A – this is equivalent to saying OUT 256*A+n,A. The second form is OUT (C),r which is equivalent to OUT 256*B+C,r. OUT sends numbers out of the Z8Ø chip and into the hardware outside. It has absolutely no effect on the flags.

OUTD OUT with Decrement. Equivalent to OUT (C),(HL) followed by DEC HL followed by DEC B. The carry flag is unchanged, but the zero flag reflects the new value of B.

OTDR A slightly different spelling in no way alters the fact that this is still an OUT with Decrement and Repeat instruction – all it does is lead us to digress from alphabetical order in order to maintain consistency. Equivalent to OUTD repeated over and over again

until B reaches zero.

OUTI As OUTD, except that HL is incremented instead of decremented.

OTIR As OTDR, except that HL is incremented instead of decremented.

POP Removes two bytes of data from the top of the stack and loads them into a register pair. Register pairs BC, DE, HL, IX and IY may be used. In addition, the instruction POP AF may be used forming a pseudo "register pair" from the accumulator and the flags register. Specifically POP will remove the topmost byte from the stack into the low part of the register pair and the next byte into the high part. The stack pointer SP is updated automatically.

PUSH PUSH is the opposite of POP. It stores the contents of any register pair at the top of the stack. The high part is pushed first, then the low part. It "remembers" that a new item has been stacked by updating the value of SP. After a PUSH instruction, SP will always point to the low part of the topmost item on the stack.

RES With this instruction we can actually alter individual bits of any register. RES is short for RESet, which means "change to zero" in computing circles, so RES is the instruction which changes any required bit of a register to zero. For instance, to reset bit 3 of D you just have to say RES 3,D. RES has no effect on any of the flags.

RET RET is used to return from a subroutine. It works by POPping an address from the stack and then jumping to that address. It is possible to alter the address to which a subroutine will return by altering the value at the top of the stack. For example, POP HL/INC HL/PUSH HL will increase the return address by one. You could, for instance, store one byte of data immediately after the CALL instruction, then POP HL/LD A,(HL)/INC HL/PUSH HL will

store that byte in A while at the same time ensuring that the subroutine will return to the address **after** that data. Another trick is to push an "artificial" return address onto the stack and then JP (or JR) to a subroutine instead of CALLing it. Now it will "return" to wherever you want it to go! Return may be used with conditions if needed. It does not alter the flags.

RL The form of this instruction is RL r. Each bit of the register specified is moved one position to the left. The leftmost bit is moved into the carry, and the rightmost bit takes on the previous value of the carry. Hence, Rotate Left. For example, if B contained 1Ø01Ø1Ø1 and the carry contained zero then RL B would leave B containing ØØ1Ø1Ø1Ø and the carry containing one. RL alters all of the flags.

RLA Note that there is no space between the L and the A. RLA is a more efficient way of doing RL A! The instruction is one byte shorter and only the carry flag is affected by this instruction.

RLC Rotate Left without Carry. RLC r is almost the same as RL r, in fact, in the sense that each bit of the register in question is moved one position to the left. Here, however, the former leftmost bit becomes both the new value of CARRY and the new rightmost bit. The former value of CARRY does not enter into the process at all. All of the flags are changed.

RLCA In one byte instead of two, RLCA is just RLC A only quicker. Only the carry flag is changed by this operation.

RLD Now for a weird one. RLD is not to be confused with RL D for it is a completely different instruction which works as follows: write the value of A and the contents of address (HL) in hex. The second hex digit of (HL) is shifted left so that it becomes the new first digit. The former value of this first digit overwrites the second digit of A, which in turn becomes the second digit of (HL). Thus, if we start off with A containing 25 and (HL) containing A3 then RLD will change things to A=2A, (HL)=35. RLD incidentally, for some reason

known only to the boffins above, is an abbreviation for Rotate Left Decimal.

RR As RL, except that the bits are moved right instead of left.

RRA As RLA, except that the bits are moved right instead of left.

RRC As RLC, except that the bits are moved right instead of left.

RRCA As RLCA, except that the bits are moved right instead of left.

RRD As RLD, except that the hex digits are moved right instead of left.

RST The same as CALL except that the instruction is but one byte long **altogether!** It is much less powerful though for two reasons: (i) you may **not** use conditions (eg RST 1Ø is legal but RST NZ,1Ø is not); and (ii) only one of eight specific addresses may be called. These are ØØ,Ø8, 1Ø,18,2Ø,28,3Ø and 38. Since the Amstrad begins executing the ROM from address ØØØØ onwards, RST ØØ is the same as switching your machine on and off.

SBC SBC, like ADC, comes in two form. The first is SBC A,r, which will first of all subtract r from A, and will then subtract the carry. Similarly SBC HL,s will subtract both s and the carry flag from HL. SBC A,A is quite a useful instruction – it leaves the carry unchanged but alters A to ØØ if there is no carry or FF if there is a carry.

SCF Set the Carry Flag. All other flags are unchanged.

SET The opposite of RES. SET 4,H will set bit 4 of register H to one (for example). Any bit of any register may be set.

SLA Shift Left Arithmetic. The form of this instruction is SLA r. It is similar to RL r except that the rightmost bit is always replaced by zero. SLA r will multiply the register r by two.

SRA Shift Right Arithmetic. Any register may be shifted right using the format SRA r. The instruction is similar to RR r except that the leftmost bit remains unchanged. SRA r will divide the register r by two if that register contains a number which is to be regarded as being in two's complement form.

SRL Shift Right Logical. Again, similar to RR except that here the leftmost bit is replaced by zero always. SRL r will divide the register r by two if that register contains a number which is to be regarded as being in absolute value form.

SUB Written as SUB r (but sometimes as SUB A,r just to be confusing). This instruction will subtract r from the register A. Note that unlike with ADD there is no corresponding instruction SUB HL,s. If you wish to subtract s from HL you must first of all reset the carry flag (usually by the use of the instruction AND A) and then use SBC HL,s.

XOR XOR r changes the value of A bit by bit. If any given bit of A is identical to the corresponding bit of r then that bit of A is reset to zero, otherwise that bit of A will be set. XOR alters all the flags and, in particular, the carry flag is always reset. Note that XOR A is the same as LD A,ØØ (ignoring flags) and that XOR FF is the same as CPL (also ignoring flags).

Chapter Nine
Logical Operators and Manipulating Bits

In this chapter, we'll cover a fairly large number of instructions which all come under the headings of either being logical operators (AND, OR, XOR), or are instructions for manipulating bits within a byte (SET, RES, BIT, Rotations and Shifts).

Let's first look at the logical operators. They are also known as **Boolean Operations.** Each operation will be described along with its **Truth Table.** This is a table showing the result of the logical operation.

AND's truth table is shown below:

A	B	A AND B
Ø	Ø	Ø
1	Ø	Ø
Ø	1	Ø
1	1	1

There are a number of things that need to be explained after looking at this table. AND works with two values. The first must be the contents of the A register, while the other can be an 8-bit number, n, or any of the single registers (including A) or the value at address IX, IY or HL. We have used register B above as an example.

Because A must always be the first of the two values, the op code for

AND A,C is written as AND C for it is assumed that A comes before the C.

The truth table shows the result of a AND on the bits of two registers. This table applies to all eight bits of the registers. If bit Ø of A or bit Ø of B is zero then the resultant AND will make bit Ø equal zero. If both bits are zero then the resultant AND will be zero while if both bits had a value of 1 then the resultant AND would be 1.

The results from the AND are stored in the A register. The value of the B register (or whatever register is used) is left unchanged. This is very important to remember. Here is a short routine showing AND in action.

```
3E 66      LD A, 1Ø2      (1Ø2 in binary is ØllØØllØ)
Ø6 2B      LD B, 43       (43 in binary is ØØlØlØll)
AØ         AND B
32 28 AØ   LD (41ØØØ),A
```

The routine AND's B and A (43 and 1Ø2) together which gives the below result:

```
ØllØØllØ
ØØlØlØll
ØØlØØØlØ
```

Only bits 1 and 5 were set (made equal to 1) as a result of the AND. The value of the A register is then placed into memory location 41ØØØ. To check that its value is indeed ØØlØØØlØ (a decimal value of 34), type PRINT PEEK(41ØØØ).

OR's truth table is as follows:

A	B	A OR B
Ø	Ø	Ø
1	Ø	1
Ø	1	1
1	1	1

OR works with all of the same values as AND, they are:

AND r	**(where r is an 8-bit register)**
AND n	**(where n is an 8-bit number)**
AND (HL)	**(the contents of memory location HL)**
AND (IX+d)	**(the contents of memory location IX or**
AND (IY+d)	**IY + the displacement)**

XOR's truth table is below:

A	B	A XOR B
Ø	Ø	Ø
1	Ø	1
Ø	1	1
1	1	Ø

This logical operator works with the same registers and addressing modes as OR and AND. XOR is short for Exclusive Oring and is a procedure in which a 1 is assigned as the result only if the two bits are different. Thus if both bits are Ø, or both bits are 1, then the result to be stored in the A register would be Ø.

We have discussed the logical operators. Now let's see them in action. They are capable of changing specific bits easily and the routine below makes use of this. You could call it a "sentence starter" or "capital letter creator". The routine takes the supplied string and changes the first character of that string to an upper case letter.

How is this done? The upper case character set starts 32 ASCII codes below the lower case set. The program takes the character code of the first character and using the A register which contains a value of 223 and the instruction AND, effectively cancels the sixth bit, making it equal to zero thus subtracting 32 from the code value. The character code is then returned. Let's look at this in binary.

A = 11Ø11111
B = Ø11????? **(we don't know the exact value of B, all we know is that it starts from 97 onwards but the vital 6th bit is always set, equalling 32 in decimal)**

The AND operation takes any bits that are both equal to 1 and makes the bit result 1. Any bits that do not equal this are reset (made equal to Ø). By making the sixth bit of the A register, zero, we are guaranteeing that the sixth bit will not be set and thus 32 is taken from the code value.

Assembler	Decimal
LD L,(IX+Ø)	DD 6E ØØ
LD H,(IX+1)	DD 66 Ø1
INC HL	23
LD C,(HL)	4E
LD A,223	3E DF
AND C	A1
LD A,C	79
LD (HL),A	77
RET	C9

Setting and Resetting

SET and RES (short for RESET) allow you to alter the value of specific bits within a register's 8-bit value. The format for SET and RES is:

SET/RES n, r

In this, r is any of the 8-bit general purpose registers or the 8-bit value at the location given by the HL, IX or IY registers. N is a number between Ø and 7 and this applies to which bit is being set or reset. (Set means being made equal to 1 while reset means making the bit equal to Ø.)

LD B,3 **means that the first and second bits are set and if we add the instruction**
SET 2,B **then the third bit is set (worth 4) and the B register is now worth in decimal terms 7, in binary ØØØØØ111.**

A most important point to remember, not only with SET and RES but with many other instructions discussed here, is that the bit parameters always run from Ø to 7. Therefore if you are talking about the third bit, this is bit 2 (as is the case in the above example). Remember this, as many books and magazine articles do not make this clear, and confusion is often caused as a result.

If we had the two instructions:

LD C,7
RES Ø,C

then the value in the C register would become 6 because the set first bit (bit Ø) has been reset thus taking one off the C register's value.

The instruction BIT can be used to see what value a bit is. You must

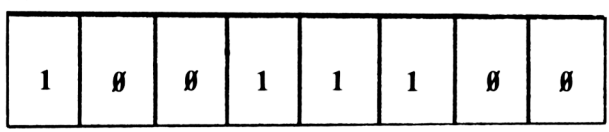
specify the bit number, and the register, in exactly the same way as with SET and RES. Its answer is not put into a register but instead into one of the flags managed by the F register. The Z or Zero flag is made equal to 1 if the bit being tested by the BIT instruction is a zero. If the bit being examined is 1 then the Zero flag is made equal to zero (the opposite way round to what you might at first think).

Rotating and Shifting

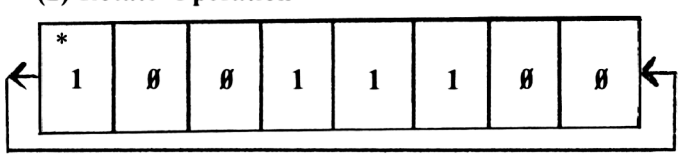
Let's now move on to rotating and shifting. The principles involved with these operations are not difficult to understand but if explained badly do become very confusing. I have enlisted the help of some diagrams to explain the subject. Let's look at the first diagram on rotating:

***Indicates bit to watch**

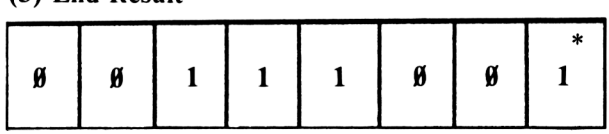
(1) Start with



(2) Rotate Operation



(3) End Result



Note that the above equals a Rotate Left with Carry (RLC)

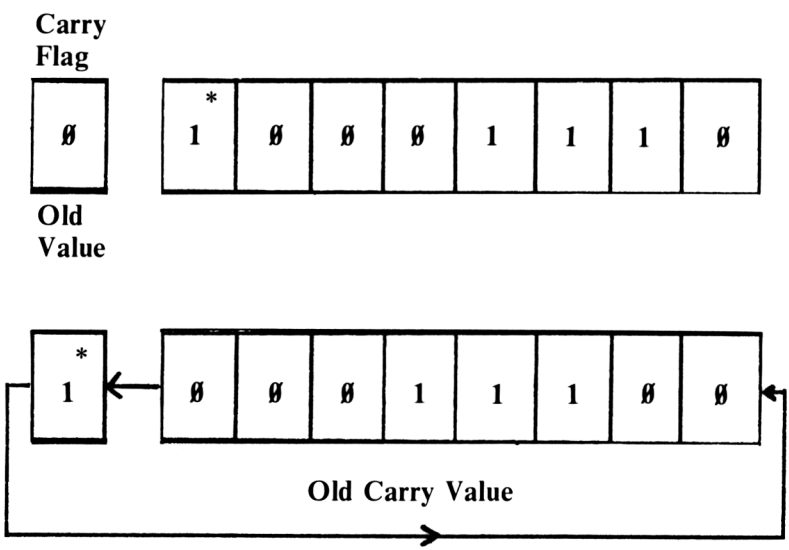
Rotate Right (RRC) is the exact opposite.

Rotating is an operation that involves a complete cycle and in this respect is different to shifting, as we will see in a moment. The end bit is taken off the byte and placed on the other end, thus moving all the other seven bits along one bit. Though the rotation shown in the diagram is to the left with all the bits moving up, this needn't always be the case.

The commands involved with this rotating are RLC and RRC. These stand for Rotate Left without Carry and Rotate Right without Carry. You can see from the diagram what this does.

There is another type of rotation and this is known as RL and RR (Rotate Left/Right **with** carry). What this does, in effect, is take the end bit (the starred one in the diagram) put it into the carry flag and take the old value from the carry flag and put that on the other end of the register. The second diagram shows this:

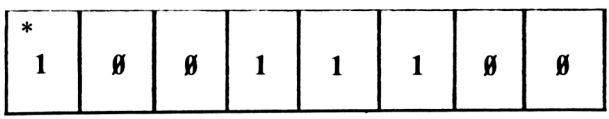
RL (Rotate Left with Carry)



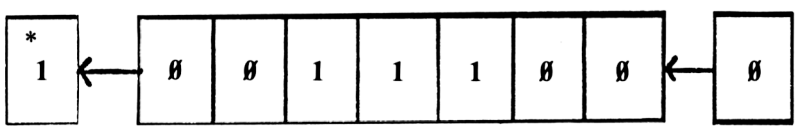
These rotation instructions work with all 8-bit registers and the values given by IX, IY and HL.

Shifting is a similar process to rotation as the third diagram shows.

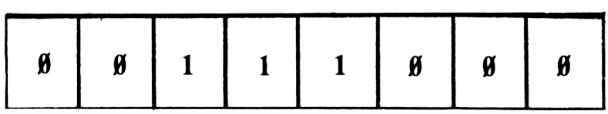
(1) Start with



(2) Shift Operation



(3) End Result



Note that this constitutes a Shift Left (known as a Shift Left Arithmetic or SLA)

There are three major shift instructions: SLA, SRA and SRL. SLA stands for Shift Left Arithmetic and is the operation featured in the diagram. SRL stands for Shift Right Logical and works in a very

similar way to SLA except it works from the right, hence bit 7 is reset and bit Ø (the rightmost bit) is pushed into carry while the rest of the bits are shifted down one bit. Thus a SRL instruction on 111ØØ1ØØ would equal Ø111ØØ1Ø.

The final shift instruction is Shift Right Arithmetic. This is similar to SRL except it shifts all the bits except bit 7 which is retained as it is. This instruction is used for when bit 7 holds the sign of the value (when used with signed integers between -127 and 127). All shifting instructions work with the same registers as the rotating instructions. What you may not have noticed, but is worth pointing out, is that SLA actually multiplies the number by 2 and SRL divides the number by 2. So to multiply a number by 8 you would use SLA three times.

Chapter Ten
Screen and ROM Routines

The screen picture displayed by your Amstrad takes up a hefty 16K (16384 bytes) in all modes. There is nothing you can do about this. (You may wish to access the screen directly in some of your machine code programs rather than using the ROM routines available. This has the advantage of making programs run much faster in most cases. Unfortunately, associating a memory address with a position on the Amstrad's screen may seem an impossible task at first due to the way the screen is stored. As well, one pixel on the screen does not necessarily relate to one bit of a byte depending on which screen mode you are using, although this is the case in mode 2. Hence, there are only two colours available in this mode as a bit can be in one of two states, set or reset.)

We will now look at how the screen is stored, assuming that it is stored from address &CØØØ to address &FFFF. This will be the case unless a machine code program has moved it somewhere else. We will assume that the offset is equal to zero. This is **not** the case if you scroll the screen.

Two Hundred High

The screen is always 2ØØ pixel lines high by 8Ø bytes across. When you set the screen mode you are telling the computer how to use these 8Ø bytes; whether you want a large number of pixels across the screen or whether you want a smaller number, using the extra bits to

store a larger range of colour values.

The screen is stored as eight 2K blocks with each block containing information on one of the eight lines of every character position. Therefore, the first 80 bytes of block one relate to the top pixel line which is line 1 of the first character row, the second 80 bytes of block one relate to line 1 of the second character row, and so on.

This means that line 1 of the bottom character row (row 25) is stored in the first block from bytes 1921 (24*80+1) to 2000 (625*80). However, each block is 2K (2048 bytes) long and so the last 48 bytes of every block are not actually used. In the same way, the second block stores information on line 2 of every character row and block eight stores information on line 8 of every character row.

If you are a bit confused, type in the following BASIC program which POKEs every byte from &C000 to &FFFF with 255. This means that it is also altering the unused bytes but the computer does not mind you doing this. Note the order in which the lines appear on the screen.

```
10 MODE 1
20 FOR n=&C000 TO &FFFF
30 POKE n,255
40 NEXT
```

Change the MODE number in line 10 and run the program again. Notice that it takes the same time to fill the screen in all modes and that different colours appear on the screen for each mode.

There are a large number of ROM routines used by the interpreter for a whole host of tasks such as printing and initialising. (These are all documented in the firmware manual available from Amsoft.)

Here are the addresses and names of a few of the more useful routines for the newer machine code programmer. Note all of the addresses are given in hexadecimal.

CALL Address	Description
BBØØ	Resets the keyboard buffer.
BB18	Waits for keypress.
BB24	Gets joystick value, number equals which bit is set: Ø=up 1=down 2=left 3=right 4=fire2 5=fire1 6=unused 7=reset
BB5A	Print character on the screen.
BBEA	Plot pixel with DE = X co-ordinate and HL = Y co-ordinate.
BBF6	Draw line from current cursor position to X, Y (held in the same register pairs as above).
BC38	Set border colours to B,C registers.
BC3E	Set flash periods H,L registers.
BC68	Set the cassette write speed.
BCAA	Add a sound to the sound queue.
BCB6	Halt all sounds.
BCD1	Introduce an RSX to the Firmware.
BD31	Send a character to the Centronics port (usually for the printer).
BB39	Set key repeat or not.
BB21	Get Caps Lock and Shift Lock states.

Pre-Packaged Machine Code Routines

One - Read a Character
Two - Rotate Left
Three - Rotate Right
Four - Big Print
Five - Massive Print
Six - Screen Fill
Seven - Headerless Load/Save
Eight - Interrupt-Driven Music
Nine - Machine Code Monitor
Ten - Box Scrolls
Eleven - RSX Chords
Twelve - Screen Compactors
Thirteen - DEEK and DOKE
Fourteen - Games Writing Package

One – Read a Character

If many BASIC programmers were asked what one additional command they would like to see on the Amstrad 464, we wouldn't be surprised if the majority asked for a SCREEN$ command. To the uninitiated, SCREEN$ and similar commands tell the programmer what character is at a specified screen location. Thus if the programmer wanted to write a routine that involved having to check the centre of the screen to see if an asterisk ('*') was present, he may have a line such as this:

IF SCREEN$(2Ø,12)=42 THEN...

The two numbers within brackets are the co-ordinates of a cursor position and the 42 is the ASCII code for an asterisk.

Unfortunately, the 464 does not have such a BASIC command and this makes life much more difficult for BASIC programmers wishing to write moving graphics games. The following routine solves this problem and will work in any of the Amstrad's three screen modes.

Type in this program and run it:

```
1 '              READ A CHARACTER
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 240
20 FOR n=40000 TO 40019
30 READ a$:POKE n,VAL("&"+a$)
```

```
40 NEXT
50 DATA DD,6E,02,DD,66,04,CD,75,BB,CD
60 DATA 60,BB,DD,6E,00,DD,66,01,77,C9
```

Type in the code and run. Then when you wish to check a specified screen position, use the following command:

CHAR%=0:CALL 40000, X, Y, @CHAR%

Here, X and Y are the co-ordinates of the position to be checked (in the same format as the BASIC LOCATE command). CHAR% is a specially created variable which, once the CALL has been executed, will contain the ASCII code of the character holding the specified position. If CHAR% is not defined in advance, an **Improper Argument** error will occur.

The CHAR variable must be an integer and can either be defined with a percentage sign following it, as above, or can be preceded by the command DEFINT C.

This is one of the shortest routines in the book and this is due to the Amstrad's ROM more than anything else. It contains a "read characters" routine which just can't be accessed by BASIC. Our program simply utilises this routine and passes the correct parameters to and from the routine.

This is what the routine looks like when disassembled:

```
                0001 ;          READ CHAR
                0002 ;
9C40            0010            ORG  40000
BB75            0020 CURSOR     DEFL 0BB75H
```

```
BB60             0030 RDCHAR DEFL  0BB60H
9C40 DD6E02      0040        LD    L,(IX+2)
9C43 DD6604      0050        LD    H,(IX+4)
9C46 CD75BB      0060        CALL  CURSOR
9C49 CD60BB      0070        CALL  RDCHAR
9C4C DD6E00      0080        LD    L,(IX+0)
9C4F DD6601      0090        LD    H,(IX+1)
9C52 77          0100        LD    (HL),A
9C53 C9          0110        RET
                 0120        END
```

Two – Rotate Left

This routine rotates a string of characters 90 degrees to the left, thus turning them on their side and printing them up the screen.

```
1 '                ROTATE LEFT
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 240
20 FOR n=40000 TO 40091
30 READ a$:POKE n,VAL("&"+a$)
40 NEXT
50 DATA DD,6E,00,DD,66,01,7E,B7,C8,23
60 DATA 5E,23,56,DD,6E,02,DD,66,04,47
70 DATA C5,E5,CD,75,BB,1A,CD,64,9C,13
80 DATA E1,2D,C1,10,F1,C9,D5,CD,A5,BB
90 DATA EB,CD,AE,BB,06,07,23,10,FD,F5
100 DATA CD,06,B9,0E,08,1A,13,D5,E5,16
110 DATA 80,1E,80,06,08,82,CB,16,2B,B3
120 DATA CB,3A,CB,2B,CB,F3,10,F3,E1,D1
130 DATA 0D,20,E4,CD,09,B9,F1,CD,5A,BB
140 DATA D1,C9
```

The format for the command is:

CALL **40000**, X, Y, @A$

X and Y are the co-ordinates of where the string is going to start being printed from and A$ equals the string to be printed which must be preceded by an '@' character. This enables the string to be passed from BASIC to machine code. The A$ is being used here to mean any string (in the same way that X can, in fact, be any numeric variable or value.) X$ and pp$ will work but 'HELLO' won't. In other words, it must be a string variable.

The X and Y co-ordinates are the same as in the BASIC command LOCATE; Y can be any row number from 1 to 25 and X can be any column number between 1 and 20 in mode 0, 40 in mode 1, and 80 in mode 2.

The routine in effect takes the first character in the string, tips it on its side and places that character into the first UDG character available (in the BASIC loader, that is character number 240). It then prints the UDG character. This makes the first available UDG character not that suitable to use when incorporating this routine into a program. To overcome this, if needs be, you can simply define SYMBOL AFTER a little lower.

```
                0001 ;          ROT   LEFT
                0002 ;
9C40            0010            ORG   40000
BB75            0020 CURSOR     DEFL  0BB75H
9C40 DD6E00     0030            LD    L,(IX+0)
9C43 DD6601     0040            LD    H,(IX+1)
9C46 7E         0050            LD    A,(HL)
9C47 B7         0060            OR    A
9C48 C8         0070            RET   Z
9C49 23         0080            INC   HL
9C4A 5E         0090            LD    E,(HL)
```

```
9C4B 23        0100          INC   HL
9C4C 56        0110          LD    D,(HL)
9C4D DD6E02    0120          LD    L,(IX+2)
9C50 DD6604    0130          LD    H,(IX+4)
9C53 47        0140          LD    B,A
9C54 C5        0150 NXTCHR   PUSH  BC
9C55 E5        0160          PUSH  HL
9C56 CD75BB    0170          CALL  CURSOR
9C59 1A        0180          LD    A,(DE)
9C5A CD649C    0190          CALL  ROTCHR
9C5D 13        0200          INC   DE
9C5E E1        0210          POP   HL
9C5F 2D        0220          DEC   L
9C60 C1        0230          POP   BC
9C61 10F1      0240          DJNZ  NXTCHR
9C63 C9        0250          RET
9C64 D5        0260 ROTCHR   PUSH  DE
BBA5           0270 MATRIX   DEFL  0BBA5H
BBAE           0280 TABLE    DEFL  0BBAEH
B906           0290 ROMENA   DEFL  0B906H
B909           0300 ROMDIS   DEFL  0B909H
BB5A           0310 TXTOUT   DEFL  0BB5AH
9C65 CDA5BB    0320          CALL  MATRIX
9C68 EB        0330          EX    DE,HL
9C69 CDAEBB    0340          CALL  TABLE
9C6C 0607      0350          LD    B,7
9C6E 23        0360 ADD7     INC   HL
```

```
9C6F 10FD      0370          DJNZ ADD7
9C71 F5        0380          PUSH AF
9C72 CD06B9    0390          CALL ROMENA
9C75 0E08      0400          LD   C,8
9C77 1A        0410 NXTBYT   LD   A,(DE)
9C78 13        0420          INC  DE
9C79 D5        0430          PUSH DE
9C7A E5        0440          PUSH HL
9C7B 1680      0450          LD   D,128
9C7D 1E80      0460          LD   E,128
9C7F 0608      0470          LD   B,8
9C81 82        0480 NXTBIT   ADD  A,D
9C82 CB16      0490          RL   (HL)
9C84 2B        0500          DEC  HL
9C85 B3        0510          OR   E
9C86 CB3A      0520          SRL  D
9C88 CB2B      0530          SRA  E
9C8A CBF3      0540          SET  6,E
9C8C 10F3      0550          DJNZ NXTBIT
9C8E E1        0560          POP  HL
9C8F D1        0570          POP  DE
9C90 0D        0580          DEC  C
9C91 20E4      0590          JR   NZ,NXTBYT
9C93 CD09B9    0600          CALL ROMDIS
9C96 F1        0610          POP  AF
9C97 CD5ABB    0620          CALL TXTOUT
```

```
9C9A D1          0630          POP   DE
9C9B C9          0640          RET
                 0650          END
```

Three – Rotate Right

This is more or less the opposite of the previous routine. With this one, the string will be printed down the screen, 90 degrees right of normal horizontal printing and 180 degrees right of Rotate Left printing.

```
1 '                ROTATE RIGHT
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 240
20 FOR n=40000 TO 40086
30 READ a$:POKE n,VAL("&"+a$)
40 NEXT
50 DATA DD,6E,00,DD,66,01,7E,B7,C8,23
60 DATA 5E,23,56,DD,6E,02,DD,66,04,47
70 DATA C5,E5,CD,75,BB,1A,CD,64,9C,13
80 DATA E1,2C,C1,10,F1,C9,D5,CD,A5,BB
90 DATA EB,CD,AE,BB,F5,CD,06,B9,0E,08
100 DATA 1A,13,D5,E5,16,80,1E,80,06,08
110 DATA 82,CB,1E,23,B3,CB,3A,CB,2B,CB
120 DATA F3,10,F3,E1,D1,0D,20,E4,CD,09
130 DATA B9,F1,CD,5A,BB,D1,C9
```

The format for the call is the same as the previous one:

CALL 40000, X, Y, @A$

Here X and Y are the co-ordinates for the string's starting position, and A$ is the string that is to be printed down the screen.

The routines have much in common as one can see from the assembler listing. The UDGs are affected in the same way with Rotate Right as they were with Rotate Left.

```
                    0001 ;          ROT   RIGHT
                    0002 ;
9C40                0010            ORG   40000
BB75                0020 CURSOR     DEFL  0BB75H
9C40 DD6E00         0030            LD    L,(IX+0)
9C43 DD6601         0040            LD    H,(IX+1)
9C46 7E             0050            LD    A,(HL)
9C47 B7             0060            OR    A
9C48 C8             0070            RET   Z
9C49 23             0080            INC   HL
9C4A 5E             0090            LD    E,(HL)
9C4B 23             0100            INC   HL
9C4C 56             0110            LD    D,(HL)
9C4D DD6E02         0120            LD    L,(IX+2)
9C50 DD6604         0130            LD    H,(IX+4)
9C53 47             0140            LD    B,A
9C54 C5             0150 NXTCHR     PUSH  BC
9C55 E5             0160            PUSH  HL
9C56 CD75BB         0170            CALL  CURSOR
9C59 1A             0180            LD    A,(DE)
9C5A CD649C         0190            CALL  ROTCHR
```

```
9C5D 13       0200          INC   DE
9C5E E1       0210          POP   HL
9C5F 2C       0220          INC   L
9C60 C1       0230          POP   BC
9C61 10F1     0240          DJNZ  NXTCHR
9C63 C9       0250          RET
9C64 D5       0260 ROTCHR   PUSH  DE
BBA5          0270 MATRIX   DEFL  0BBA5H
BBAE          0280 TABLE    DEFL  0BBAEH
B906          0290 ROMENA   DEFL  0B906H
B909          0300 ROMDIS   DEFL  0B909H
BB5A          0310 TXTOUT   DEFL  0BB5AH
9C65 CDA5BB   0320          CALL  MATRIX
9C68 EB       0330          EX    DE,HL
9C69 CDAEBB   0340          CALL  TABLE
9C6C F5       0350          PUSH  AF
9C6D CD06B9   0360          CALL  ROMENA
9C70 0E08     0370          LD    C,8
9C72 1A       0380 NXTBYT   LD    A,(DE)
9C73 13       0390          INC   DE
9C74 D5       0400          PUSH  DE
9C75 E5       0410          PUSH  HL
9C76 1680     0420          LD    D,128
9C78 1E80     0430          LD    E,128
9C7A 0608     0440          LD    B,8
9C7C 82       0450 NXTBIT   ADD   A,D
9C7D CB1E     0460          RR    (HL)
```

```
9C7F 23          0470          INC   HL
9C80 B3          0480          OR    E
9C81 CB3A        0490          SRL   D
9C83 CB2B        0500          SRA   E
9C85 CBF3        0510          SET   6,E
9C87 10F3        0520          DJNZ  NXTBIT
9C89 E1          0530          POP   HL
9C8A D1          0540          POP   DE
9C8B 0D          0550          DEC   C
9C8C 20E4        0560          JR    NZ,NXTBYT
9C8E CD09B9      0570          CALL  ROMDIS
9C91 F1          0580          POP   AF
9C92 CD5ABB      0590          CALL  TXTOUT
9C95 D1          0600          POP   DE
9C96 C9          0610          RET
                 0620          END
```

Four – Big Print

Double-size characters are now possible with this routine. There are many obvious uses for such a routine, one being for titles of programs or screen displays.

The BASIC loader should be typed in as normal, saved onto tape or cassette and then run.

```
1 '                BIG PRINT
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 240
20 FOR n=40000 TO 40159
30 READ a$:POKE n,VAL("&"+a$)
40 NEXT
50 DATA CD,93,BB,F5,DD,6E,04,DD,66,05
60 DATA 46,23,5E,23,56,DD,6E,06,DD,66
70 DATA 08,C5,D5,E5,1A,47,CD,06,B9,78
80 DATA CD,9D,9C,47,CD,09,B9,E1,5D,54
90 DATA CD,75,BB,DD,7E,02,CD,90,BB,78
100 DATA CD,5A,BB,3C,CD,5A,BB,6B,62,2C
110 DATA CD,75,BB,DD,7E,00,CD,90,BB,78
120 DATA 3C,3C,CD,5A,BB,3C,CD,5A,BB,6B
130 DATA 62,24,24,D1,13,C1,10,BD,F1,CD
140 DATA 90,BB,C9,CD,A5,BB,EB,CD,AE,BB
```

```
150 DATA F5,0E,02,06,04,C5,1A,0F,0F,0F
160 DATA 0F,06,04,1F,CB,1E,CB,2E,10,F9
170 DATA 7E,23,77,06,07,23,10,FD,1A,06
180 DATA 04,1F,CB,1E,CB,2E,10,F9,7E,23
190 DATA 77,06,07,2B,10,FD,13,C1,10,D3
200 DATA 06,08,23,10,FD,0D,20,C9,F1,C9
```

To use the routine, you need the following CALL command:

CALL 40000, X, Y, @A$, P1, P2

By now you should be able to work out the first two or three parameters for yourself. In case you don't know, or you've been very naughty and flicked through the book, reading at odd points, then I'll tell you. X and Y are the top left co-ordinates. The double size characters are printed from here. A$ is the string containing the characters that are to be printed. The particular variables used in the example need not be the ones that you use in your program; any numeric or string variable may be used and for the co-ordinates, ordinary numbers can be utilised.

This is also the case for the two final parameters which I've given the names P1 and P2. They can use either numeric variables or ordinary numbers and they set the colour of the characters. P1 controls the top half of the character and P2 the bottom half. These pen variables can be set to **any** number. This is because, if the parameter is out of the range of the usual values (shown in the table below), then the routine will mask off enough bits to make them fit within the normal pen colour range.

Normal Values:	**Mode**	**Range**
	0	**0-15**
	1	**0-3**
	2	**0-1**

The routine sets the current pen colours back to what they were before the routine was called. This routine utilises the first four UDG characters to create the double-size characters. Therefore make sure that there are at least four UDG characters available. (If you are not using any UDGs then leave the SYMBOL AFTER 240 command, found in the BASIC loader, in your program.)

```
                    0001 ;        BIG   PRINT
                    0002 ;
9C40                0010          ORG   40000
B906                0020 ROMENA   DEFL  0B906H
B909                0030 ROMDIS   DEFL  0B909H
BBA5                0040 MATRIX   DEFL  0BBA5H
BBAE                0050 TABLE    DEFL  0BBAEH
BB75                0060 CURSOR   DEFL  0BB75H
BB93                0070 GETPEN   DEFL  0BB93H
BB90                0080 SETPEN   DEFL  0BB90H
BB5A                0090 TXTOUT   DEFL  0BB5AH
9C40 CD93BB         0100          CALL  GETPEN
9C43 F5             0110          PUSH  AF
9C44 DD6E04         0120          LD    L,(IX+4)
9C47 DD6605         0130          LD    H,(IX+5)
9C4A 46             0140          LD    B,(HL)
9C4B 23             0150          INC   HL
9C4C 5E             0160          LD    E,(HL)
9C4D 23             0170          INC   HL
9C4E 56             0180          LD    D,(HL)
9C4F DD6E06         0190          LD    L,(IX+6)
```

```
9C52 DD6608    0200          LD    H,(IX+8)
9C55 C5        0210 NXTCHR   PUSH  BC
9C56 D5        0220          PUSH  DE
9C57 E5        0230          PUSH  HL
9C58 1A        0240          LD    A,(DE)
9C59 47        0250          LD    B,A
9C5A CD06B9    0260          CALL  ROMENA
9C5D 78        0270          LD    A,B
9C5E CD9D9C    0280          CALL  BIGCHR
9C61 47        0290          LD    B,A
9C62 CD09B9    0300          CALL  ROMDIS
9C65 E1        0310          POP   HL
9C66 5D        0320          LD    E,L
9C67 54        0330          LD    D,H
9C68 CD75BB    0340          CALL  CURSOR
9C6B DD7E02    0350          LD    A,(IX+2)
9C6E CD90BB    0360          CALL  SETPEN
9C71 78        0370          LD    A,B
9C72 CD5ABB    0380          CALL  TXTOUT
9C75 3C        0390          INC   A
9C76 CD5ABB    0400          CALL  TXTOUT
9C79 6B        0410          LD    L,E
9C7A 62        0420          LD    H,D
9C7B 2C        0430          INC   L
9C7C CD75BB    0440          CALL  CURSOR
9C7F DD7E00    0450          LD    A,(IX+0)
9C82 CD90BB    0460          CALL  SETPEN
```

```
9C85 78       0470         LD    A,B
9C86 3C       0480         INC   A
9C87 3C       0490         INC   A
9C88 CD5ABB   0500         CALL  TXTOUT
9C8B 3C       0510         INC   A
9C8C CD5ABB   0520         CALL  TXTOUT
9C8F 6B       0530         LD    L,E
9C90 62       0540         LD    H,D
9C91 24       0550         INC   H
9C92 24       0560         INC   H
9C93 D1       0570         POP   DE
9C94 13       0580         INC   DE
9C95 C1       0590         POP   BC
9C96 10BD     0600         DJNZ  NXTCHR
9C98 F1       0610         POP   AF
9C99 CD90BB   0620         CALL  SETPEN
9C9C C9       0630         RET
9C9D CDA5BB   0640 BIGCHR  CALL  MATRIX
9CA0 EB       0650         EX    DE,HL
9CA1 CDAEBB   0660         CALL  TABLE
9CA4 F5       0670         PUSH  AF
9CA5 0E02     0680         LD    C,2
9CA7 0604     0690 NXTSET  LD    B,4
9CA9 C5       0700 NXTROW  PUSH  BC
9CAA 1A       0710         LD    A,(DE)
9CAB 0F       0720         RRCA
9CAC 0F       0730         RRCA
```

```
9CAD 0F        0740          RRCA
9CAE 0F        0750          RRCA
9CAF 0604      0760          LD    B,4
9CB1 1F        0770 LFTBYT   RRA
9CB2 CB1E      0780          RR    (HL)
9CB4 CB2E      0790          SRA   (HL)
9CB6 10F9      0800          DJNZ  LFTBYT
9CB8 7E        0810          LD    A,(HL)
9CB9 23        0820          INC   HL
9CBA 77        0830          LD    (HL),A
9CBB 0607      0840          LD    B,7
9CBD 23        0850 NXT1     INC   HL
9CBE 10FD      0860          DJNZ  NXT1
9CC0 1A        0870          LD    A,(DE)
9CC1 0604      0880          LD    B,4
9CC3 1F        0890 RGTBYT   RRA
9CC4 CB1E      0900          RR    (HL)
9CC6 CB2E      0910          SRA   (HL)
9CC8 10F9      0920          DJNZ  RGTBYT
9CCA 7E        0930          LD    A,(HL)
9CCB 23        0940          INC   HL
9CCC 77        0950          LD    (HL),A
9CCD 0607      0960          LD    B,7
9CCF 2B        0970 NXT2     DEC   HL
9CD0 10FD      0980          DJNZ  NXT2
9CD2 13        0990          INC   DE
9CD3 C1        1000          POP   BC
```

```
9CD4 10D3      1010          DJNZ NXTROW
9CD6 0608      1020          LD   B,8
9CD8 23        1030 NXT3     INC  HL
9CD9 10FD      1040          DJNZ NXT3
9CDB 0D        1050          DEC  C
9CDC 20C9      1060          JR   NZ,NXTSET
9CDE F1        1070          POP  AF
9CDF C9        1080          RET
               1090          END
```

Five – Massive Print

Massive Print performs character enlargement but on a far bigger scale than **Big Print.** Each character when reproduced on the screen is 16 times bigger than normal.

If you glance at the two routines, **Big Print** and **Massive Print,** you will see many differences. In fact, the two routines are completely different. **Massive Print** was far easier to write than **Big Print.** The **Massive Print** routine assigns a two by two block of CHR$(143)'s (solid blocks) for each pixel of a normal character whereas the **Big Print** routine cannot use this time-saving method of character enlargement and must rely on a more complex and longer routine to work.

```
1 '              MASSIVE PRINT
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 240
20 FOR n=40000 TO 40085
30 READ a$:POKE n,VAL("&"+a$)
40 NEXT
50 DATA CD,93,BB,F5,CD,06,B9,DD,7E,04
60 DATA CD,A5,BB,EB,DD,6E,06,DD,66,08
70 DATA 06,08,C5,0E,80,06,02,C5,E5,CD
80 DATA 75,BB,CB,40,28,05,DD,7E,00,18
90 DATA 03,DD,7E,02,CD,90,BB,E1,06,08
```

```
100 DATA 1A,A1,28,04,3E,8F,18,02,3E,20
110 DATA CD,5A,BB,CD,5A,BB,CB,39,10,EC
120 DATA 2C,C1,10,D1,13,C1,10,C8,CD,09
130 DATA B9,F1,CD,90,BB,C9
```

The format for the CALL is:

CALL 40000, X, Y, CHAR, P1, P2

The CALL parameters are very similar to the previous large printing routine with one exception. Whereas **Big Print** will print a string of characters, **Massive Print** with its far larger size asks for the ASCII code of the one character which it will enlarge. If you wish to disply more than one enlarged character on the screen, call the routine once for every character.

The routine will work in all three of the screen modes and uses the two pen colour parameters in the same way as **Big Print.** If, of course, you want single colour characters, make P1 and P2 the same number.

```
                0001 ;          MSVE PRINT
                0002 ;
9C40            0010            ORG  40000
BB93            0020 GETPEN     DEFL 0BB93H
BB90            0030 SETPEN     DEFL 0BB90H
BBA5            0040 MATRIX     DEFL 0BBA5H
BB75            0050 CURSOR     DEFL 0BB75H
BB5A            0060 TXTOUT     DEFL 0BB5AH
```

```
B906             0070 ROMENA DEFL 0B906H
B909             0080 ROMDIS DEFL 0B909H
9C40 CD93BB      0090        CALL GETPEN
9C43 F5          0100        PUSH AF
9C44 CD06B9      0110        CALL ROMENA
9C47 DD7E04      0120        LD   A,(IX+4)
9C4A CDA5BB      0130        CALL MATRIX
9C4D EB          0140        EX   DE,HL
9C4E DD6E06      0150        LD   L,(IX+6)
9C51 DD6608      0160        LD   H,(IX+8)
9C54 0608        0170        LD   B,8
9C56 C5          0180 NXTROW PUSH BC
9C57 0E80        0190        LD   C,128
9C59 0602        0200        LD   B,2
9C5B C5          0210 AGAIN  PUSH BC
9C5C E5          0220        PUSH HL
9C5D CD75BB      0230        CALL CURSOR
9C60 CB40        0240        BIT  0,B
9C62 2805        0250        JR   Z,COL2
9C64 DD7E00      0260        LD   A,(IX+0)
9C67 1803        0270        JR   SETCOL
9C69 DD7E02      0280 COL2   LD   A,(IX+2)
9C6C CD90BB      0290 SETCOL CALL SETPEN
9C6F E1          0300        POP  HL
9C70 0608        0310        LD   B,8
9C72 1A          0320 NXTCOL LD   A,(DE)
9C73 A1          0330        AND  C
```

```
9C74 2804      0340         JR    Z,SPACE
9C76 3E8F      0350         LD    A,143
9C78 1802      0360         JR    PRINT
9C7A 3E20      0370 SPACE   LD    A,32
9C7C CD5ABB    0380 PRINT   CALL  TXTOUT
9C7F CD5ABB    0390         CALL  TXTOUT
9C82 CB39      0400         SRL   C
9C84 10EC      0410         DJNZ  NXTCOL
9C86 2C        0420         INC   L
9C87 C1        0430         POP   BC
9C88 10D1      0440         DJNZ  AGAIN
9C8A 13        0450         INC   DE
9C8B C1        0460         POP   BC
9C8C 10C8      0470         DJNZ  NXTROW
9C8E CD09B9    0480         CALL  ROMDIS
9C91 F1        0490         POP   AF
9C92 CD90BB    0500         CALL  SETPEN
9C95 C9        0510         RET
               0520         END
```

Six – Screen Fill

This simple routine fills the Amstrad's screen with whatever ASCII characters you require.

```
1 '            SCREEN FILL
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 240
20 FOR n=40000 TO 40021
30 READ a$:POKE n,VAL("&"+a$)
40 NEXT
50 DATA DD,7E,00,01,E8,03,F5,C5,CD,5A
60 DATA BB,C1,0B,78,B1,28,03,F1,18,F2
70 DATA F1,C9
```

The format for the call is straightforward:

CALL 40000,A

A is simply the ASCII code of the character you wish to print on-screen and A can be a variable or can be replaced by a normal number. Hence CALL 40000,66 will fill the screen with B's. The routine, at present, will only work with the MODE 1 screen.

Here is an ideal opportunity for you to delve into a little alteration work. If you look at the assembler listing of this routine you can see

that line 40 has the instruction LD BC,1000. This loading of the register BC with the value 1000 is telling the routine how many screen spaces there are to fill with a character. In mode 1, this is 40 x 25 = 1000. Mode 0 is only 500 (20 x 25) and mode 2 twice as many (80 x 25 = 2000). Therefore altering BC to 500 will enable it to print in mode 0 just filling one complete screen.

```
                    0001 ;        SCREEN FILL
                    0002 ;
9C40                0010          ORG   40000
BB5A                0020 PRNT     DEFL  0BB5AH
9C40 DD7E00         0030          LD    A,(IX+0)
9C43 01E803         0040          LD    BC,1000
9C46 F5             0050 LOOP     PUSH  AF
9C47 C5             0060          PUSH  BC
9C48 CD5ABB         0070          CALL  PRNT
9C4B C1             0080          POP   BC
9C4C 0B             0090          DEC   BC
9C4D 78             0100          LD    A,B
9C4E B1             0110          OR    C
9C4F 2803           0120          JR    Z,END
9C51 F1             0130          POP   AF
9C52 18F2           0140          JR    LOOP
9C54 F1             0150 END      POP   AF
9C55 C9             0160          RET
                    0170          END
```

If you have an assembler, it is just a matter of editing line 40 and altering it to LD BC, 500 but if you don't then it's a fraction more

complicated. The three hex pairs to the left of the line number 40 are the equivalent of the op code LD BC,1000. We wish to alter the last two pairs which make up the number, 1000. Using the HEX$ command, PRINT HEX$(500) to give hexadecimal value, F4 01. Returning to the BASIC listing, we alter the appropriate data (the fifth and sixth numbers) to F4 01 and then re-run the program, after first saving it to tape or disc. The program should now work perfectly in mode 0. From this, it can easily be altered to work in mode 2 or to just print a few lines of a certain character in a certain mode.

Seven – Headerless Load/Save

If you have a 464, and you've watched it loading in a long program, you would have noticed that it loads in a large number of blocks. Each block is proceded by a header which contains information such as the start address and the length of the data.

This program scraps the headers, making the program load and save in just one big block and cutting down greatly on the saving and loading time needed. When called, the routine does not issue prompt messages.

```
1 '              HEADERLESS
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 240
20 FOR n=40000 TO 40166
30 READ a$:POKE n,VAL("&"+a$)
40 NEXT
50 DATA FE,03,28,16,21,C0,9C,FE,02,20
60 DATA 5F,AF,DD,5E,00,DD,56,01,DD,6E
70 DATA 02,DD,66,03,18,0F,DD,7E,00,DD
80 DATA 5E,02,DD,56,03,DD,6E,04,DD,66
90 DATA 05,CD,9E,BC,30,36,C9,FE,03,28
100 DATA 16,21,C0,9C,FE,02,20,30,AF,DD
110 DATA 5E,00,DD,56,01,DD,6E,02,DD,66
```

```
120 DATA 03,18,0F,DD,7E,00,DD,5E,02,DD
130 DATA 56,03,DD,6E,04,DD,66,05,CD,A1
140 DATA BC,D8,B7,28,05,21,D3,9C,18,06
150 DATA CD,03,BB,21,DF,9C,7E,23,CB,7F
160 DATA 20,05,CD,5A,BB,18,F5,CB,BF,CD
170 DATA 5A,BB,3E,07,CD,5A,BB,C9,2A,50
180 DATA 41,52,41,4D,45,54,45,52,20,4D
190 DATA 49,53,53,49,4E,47,AA,2A,52,45
200 DATA 41,44,20,45,52,52,4F,52,AA,2A
210 DATA 45,53,43,41,50,45,AA
```

The format for this CALL is:

CALL 40000, S, L, SYNC

S is the start address and L the length of the program as there are no headers to carry this information. The SYNC variable is used by your Amstrad to distinguish the header (character 44) from the data (represented by a SYNC value of 22). As we have no headers, the value included is not important, except that the value which is used to SAVE a program, must be used to LOAD it.

```
                0001 ;              HEADERLESS
                0002 ;
9C40            0010          ORG   40000
                0020 ;        SAVE  ROUTINE
BC9E            0030 SAVE     DEFL  0BC9EH
BCA1            0040 LOAD     DEFL  0BCA1H
BB5A            0050 TXTOUT   DEFL  0BB5AH
```

```
BB03             0060 KRESET DEFL 0BB03H
9C40 FE03        0070        CP   3
9C42 2816        0080        JR   Z,SVSYNC
9C44 21C09C      0090        LD   HL,PARAM
9C47 FE02        0100        CP   2
9C49 205F        0110        JR   NZ,ERROR
9C4B AF          0120        XOR  A
9C4C DD5E00      0130        LD   E,(IX+0)
9C4F DD5601      0140        LD   D,(IX+1)
9C52 DD6E02      0150        LD   L,(IX+2)
9C55 DD6603      0160        LD   H,(IX+3)
9C58 180F        0170        JR   SVESET
9C5A DD7E00      0180 SVSYNC LD   A,(IX+0)
9C5D DD5E02      0190        LD   E,(IX+2)
9C60 DD5603      0200        LD   D,(IX+3)
9C63 DD6E04      0210        LD   L,(IX+4)
9C66 DD6605      0220        LD   H,(IX+5)
9C69 CD9EBC      0230 SVESET CALL SAVE
9C6C 3036        0240        JR   NC,ESC
9C6E C9          0250        RET
                 0260 ;
                 0270 ;      LOAD ROUTINE
9C6F FE03        0280        CP   3
9C71 2816        0290        JR   Z,LDSYNC
9C73 21C09C      0300        LD   HL,PARAM
9C76 FE02        0310        CP   2
9C78 2030        0320        JR   NZ,ERROR
```

```
9C7A AF        0330          XOR  A
9C7B DD5E00    0340          LD   E,(IX+0)
9C7E DD5601    0350          LD   D,(IX+1)
9C81 DD6E02    0360          LD   L,(IX+2)
9C84 DD6603    0370          LD   H,(IX+3)
9C87 180F      0380          JR   LDSET
9C89 DD7E00    0390 LDSYNC   LD   A,(IX+0)
9C8C DD5E02    0400          LD   E,(IX+2)
9C8F DD5603    0410          LD   D,(IX+3)
9C92 DD6E04    0420          LD   L,(IX+4)
9C95 DD6605    0430          LD   H,(IX+5)
9C98 CDA1BC    0440 LDSET    CALL LOAD
9C9B D8        0450          RET  C
9C9C B7        0460          OR   A
9C9D 2805      0470          JR   Z,ESC
9C9F 21D39C    0480          LD   HL,TAPERR
9CA2 1806      0490          JR   ERROR
9CA4 CD03BB    0500 ESC      CALL KRESET
9CA7 21DF9C    0510          LD   HL,ESCAPE
9CAA 7E        0520 ERROR    LD   A,(HL)
9CAB 23        0530          INC  HL
9CAC CB7F      0540          BIT  7,A
9CAE 2005      0550          JR   NZ,ENDERR
9CB0 CD5ABB    0560          CALL TXTOUT
9CB3 18F5      0570          JR   ERROR
9CB5 CBBF      0580 ENDERR   RES  7,A
```

```
9CB7 CD5ABB   0590         CALL  TXTOUT
9CBA 3E07     0600         LD    A,7
9CBC CD5ABB   0610         CALL  TXTOUT
9CBF C9       0620         RET
9CC0          0630 PARAM   DEFM  "*PARAMETER"
9CCA          0640         DEFM  " MISSING"
9CD2 AA       0650         DEFB  "*"+80H
9CD3          0660 TAPERR  DEFM  "*READ"
9CD8          0670         DEFM  " ERROR"
9CDE AA       0680         DEFB  "*"+80H
9CDF          0690 ESCAPE  DEFM  "*ESCAPE"
9CE6 AA       0700         DEFB  "*"+80H
              0710         END
```

Eight – Interrupt-Driven Music

Many commercial arcade games allow music to play constantly in the background while the game is played. We wanted to produce a similar effect that you could include within your games. The result is this program.

The program is one of the largest in the book. The storing of data for the tune is a byte-consuming task. Even so, the whole routine (including the tune) takes up only 59Ø bytes leaving you still with around 43K for your arcade game. The machine code routine needs to be used in conjunction with a small piece of BASIC:

```
1Ø CALL 4ØØØØ,Ø
2Ø EVERY 6 GOSUB 13ØØØ

...Rest Of Game...

13ØØØ CALL 4ØØØØ:RETURN
```

The first CALL needs to be followed by any parameter (we've chosen Ø here; it doesn't really matter which value you use). This initialises the routine and sets the music data up ready to play. The second line utilises the BASIC command EVERY for controlling system interrupts. The 6 is approximately 6/5Øths of a second. If you are not familiar with the command, EVERY or system interrupts then I suggest you look in the Amstrad manual, chapter 9.3 and chapter 1Ø. EVERY 6/5Øths of a second the program will jump to the subroutine at line 13ØØØ (though you can locate this subroutine at any place you wish). The CALL here simply plays the next note of

the tune and then returns to the normal program.

The 6/5Øths time can be altered to whatever the user requires, this is just an example. The sound uses only channel 1 and the ENV volume envelope 15. This envelope can be re-defined through BASIC to create some strange effects. The tune can be stopped by simply disabling the system interrupts. Fortunately, we have a BASIC command to take the hard work out of this, it is simply called DI (Disable Interrupts). The opposite of this command is EI (Enable Interrupts).

```
1 '           INTERRUPT MUSIC
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 240
20 FOR n=40000 TO 40590
30 READ a$:POKE n,VAL("&"+a$)
40 NEXT
50 DATA B7,28,14,21,95,9C,22,5D,9C,3E
60 DATA 01,32,84,9C,3E,0F,21,85,9C,CD
70 DATA BC,BC,C9,21,84,9C,35,C0,11,95
80 DATA 9C,1A,B7,C8,13,77,3C,20,09,1A
90 DATA 77,21,95,9C,22,5D,9C,C9,1A,6F
100 DATA 13,1A,67,13,ED,53,5D,9C,22,8F
110 DATA 9C,21,8C,9C,CD,AA,BC,C9,00,02
120 DATA 03,05,01,0F,FF,0A,81,0F,00,00
130 DATA 00,00,00,00,00
140 REM          MUSIC DATA
150 DATA 02,AA,01,02,92
160 DATA 01,01,7B,01,02,EF,00,01,7B,01
170 DATA 02,EF,00,01,7B,01,06,EF,00,01
```

```
180 DATA EF,00,01,D5,00,01,C9,00,01,BE
190 DATA 00,01,EF,00,01,D5,00,02,BE,00
200 DATA 01,FD,00,02,D5,00,06,EF,00,02
210 DATA AA,01,02,92,01,01,7B,01,02,EF
220 DATA 00,01,7B,01,02,EF,00,01,7B,01
230 DATA 06,EF,00,01,1C,01,01,3F,01,01
240 DATA 52,01,01,1C,01,01,EF,00,02,BE
250 DATA 00,01,D5,00,01,EF,00,01,1C,01
260 DATA 06,D5,00,02,AA,01,02,92,01,01
270 DATA 7B,01,02,EF,00,01,7B,01,02,EF
280 DATA 00,01,7B,01,06,EF,00,01,EF,00
290 DATA 01,D5,00,01,C9,00,01,BE,00,01
300 DATA EF,00,01,D5,00,02,BE,00,01,FD
310 DATA 00,02,D5,00,06,EF,00,01,EF,00
320 DATA 01,D5,00,01,BE,00,01,EF,00,01
330 DATA D5,00,02,BE,00,01,EF,00,01,D5
340 DATA 00,01,EF,00,01,BE,00,01,EF,00
350 DATA 01,D5,00,02,BE,00,01,EF,00,01
360 DATA D5,00,01,EF,00,01,BE,00,01,EF
370 DATA 00,01,D5,00,02,BE,00,01,FD,00
380 DATA 02,D5,00,06,EF,00,01,7B,01,01
390 DATA 66,01,01,52,01,02,3F,01,01,1C
400 DATA 01,02,3F,01,01,7B,01,01,66,01
410 DATA 01,52,01,02,3F,01,01,1C,01,02
420 DATA 3F,01,01,BE,00,01,EF,00,01,3F
430 DATA 01,01,1C,01,01,FD,00,01,EF,00
```

```
440 DATA 01,D5,00,01,BE,00,01,D5,00,01
450 DATA EF,00,01,D5,00,05,3F,01,01,3F
460 DATA 01,01,7B,01,01,66,01,02,3F,01
470 DATA 01,1C,01,02,3F,01,01,7B,01,01
480 DATA 66,01,01,52,01,02,3F,01,01,1C
490 DATA 01,02,3F,01,01,3F,01,01,1C,01
500 DATA 01,0C,01,01,FD,00,02,FD,00,02
510 DATA FD,00,01,1C,01,01,52,01,01,AA
520 DATA 01,05,3F,01,01,7B,01,01,66,01
530 DATA 01,52,01,02,3F,01,01,1C,01,02
540 DATA 3F,01,01,7B,01,01,66,01,01,52
550 DATA 01,02,3F,01,01,1C,01,02,3F,01
560 DATA 01,BE,00,01,EF,00,01,3F,01,01
570 DATA 1C,01,01,FD,00,01,EF,00,01,D5
580 DATA 00,01,BE,00,01,D5,00,01,EF,00
590 DATA 01,D5,00,05,EF,00,01,3F,01,01
600 DATA 52,01,01,3F,01,02,EF,00,01,1C
610 DATA 01,02,EF,00,01,1C,01,01,EF,00
620 DATA 01,1C,01,01,3F,01,01,EF,00,01
630 DATA BE,00,02,9F,00,01,BE,00,01,EF
640 DATA 00,01,3F,01,02,1C,01,02,EF,00
650 DATA 01,BE,00,03,D5,00,06,EF,00,FF
660 DATA 01
```

Each note of the tune consists of three bytes of data. The first byte deals with the length of the note:

1 = a semi-quaver
2 = a quaver
3 = a dotted quaver
4 = a crochet
6 = a dotted crochet
8 = a minim
12 = a dotted minim

The pitch of your note can theoretically be between 1 and 4000, though in practice you will probably use a range of between 50 and 1300. This means that we have to use two bytes in which to hold the value. Byte two holds the low byte of the tone and byte three holds the high byte. These correspond to the formula:

Pitch = Byte Two + 256 * Byte Three

The music data starts at line 150 of the BASIC loader. It's simpler to use an assembler to enter the music data, but if you don't have one you can work out the hexadecimal values for each byte, and enter them into the BASIC loader. Keep in mind that the low byte precedes the high byte of the pitch.

Two values for the length of the note have special effects. Zero constitutes the end of the data and will not play any more even if the BASIC EVERY command is still calling the subroutine. The second special value 255,n tells the program to repeat the tune again after a wait of n, where n is comparable to one of the note values (shown earlier in a table).

```
                0001 ;          INT   MUSIC
                0002 ;
9C40            0010            ORG   40000
BCBC            0020 AMPENV     DEFL  0BCBCH
BCAA            0030 ADDSND     DEFL  0BCAAH
9C40 B7         0040            OR    A
9C41 2814       0050            JR    Z,NXTSND
9C43 21959C     0060            LD    HL,MUSIC
9C46 225D9C     0070            LD    (DATADR+1),HL
9C49 3E01       0080            LD    A,1
9C4B 32849C     0090            LD    (TIME),A
9C4E 3E0F       0100            LD    A,15
9C50 21859C     0110            LD    HL,AMPL
9C53 CDBCBC     0120            CALL  AMPENV
9C56 C9         0130            RET
9C57 21849C     0140 NXTSND     LD    HL,TIME
9C5A 35         0150            DEC   (HL)
9C5B C0         0160            RET   NZ
9C5C 11959C     0170 DATADR     LD    DE,MUSIC
9C5F 1A         0180            LD    A,(DE)
9C60 B7         0190            OR    A
9C61 C8         0200            RET   Z
9C62 13         0210            INC   DE
9C63 77         0220            LD    (HL),A
9C64 3C         0230            INC   A
9C65 2009       0240            JR    NZ,CONT
9C67 1A         0250            LD    A,(DE)
```

```
9C68 77          0260          LD    (HL),A
9C69 21959C      0270          LD    HL,MUSIC
9C6C 225D9C      0280          LD    (DATADR+1),HL
9C6F C9          0290          RET
9C70 1A          0300 CONT     LD    A,(DE)
9C71 6F          0310          LD    L,A
9C72 13          0320          INC   DE
9C73 1A          0330          LD    A,(DE)
9C74 67          0340          LD    H,A
9C75 13          0350          INC   DE
9C76 ED535D9C    0360          LD    (DATADR+1),DE
9C7A 228F9C      0370          LD    (TONE),HL
9C7D 218C9C      0380          LD    HL,SOUND
9C80 CDAABC      0390          CALL  ADDSND
9C83 C9          0400          RET
9C84 00          0410 TIME     DEFB  0
9C85 02          0420 AMPL     DEFB  2,3,5,1
     03 05 01
9C89 0F          0430          DEFB  15,255,10
     FF 0A
9C8C 81          0440 SOUND    DEFB  129,15,0
     0F 00
9C8F 00          0450 TONE     DEFB  0,0,0,0,0,0
     00 00 00 00 00
9C95 02          0460 MUSIC    DEFB  2
9C96 AA01        0470          DEFW  426
```

```
9C98 02        0480        DEFB 2
9C99 9201      0490        DEFW 402
9C9B 01        0500        DEFB 1
9C9C 7B01      0510        DEFW 379
9C9E 02        0520        DEFB 2
9C9F EF00      0530        DEFW 239
9CA1 01        0540        DEFB 1
9CA2 7B01      0550        DEFW 379
9CA4 02        0560        DEFB 2
9CA5 EF00      0570        DEFW 239
9CA7 01        0580        DEFB 1
9CA8 7B01      0590        DEFW 379
9CAA 06        0600        DEFB 6
9CAB EF00      0610        DEFW 239
9CAD 01        0620        DEFB 1
9CAE EF00      0630        DEFW 239
9CB0 01        0640        DEFB 1
9CB1 D500      0650        DEFW 213
9CB3 01        0660        DEFB 1
9CB4 C900      0670        DEFW 201
9CB6 01        0680        DEFB 1
9CB7 BE00      0690        DEFW 190
9CB9 01        0700        DEFB 1
9CBA EF00      0710        DEFW 239
9CBC 01        0720        DEFB 1
9CBD D500      0730        DEFW 213
9CBF 02        0740        DEFB 2
```

```
9CC0 BE00      0750        DEFW 190
9CC2 01        0760        DEFB 1
9CC3 FD00      0770        DEFW 253
9CC5 02        0780        DEFB 2
9CC6 D500      0790        DEFW 213
9CC8 06        0800        DEFB 6
9CC9 EF00      0810        DEFW 239
9CCB 02        0820        DEFB 2
9CCC AA01      0830        DEFW 426
9CCE 02        0840        DEFB 2
9CCF 9201      0850        DEFW 402
9CD1 01        0860        DEFB 1
9CD2 7B01      0870        DEFW 379
9CD4 02        0880        DEFB 2
9CD5 EF00      0890        DEFW 239
9CD7 01        0900        DEFB 1
9CD8 7B01      0910        DEFW 379
9CDA 02        0920        DEFB 2
9CDB EF00      0930        DEFW 239
9CDD 01        0940        DEFB 1
9CDE 7B01      0950        DEFW 379
9CE0 06        0960        DEFB 6
9CE1 EF00      0970        DEFW 239
9CE3 01        0980        DEFB 1
9CE4 1C01      0990        DEFW 284
9CE6 01        1000        DEFB 1
9CE7 3F01      1010        DEFW 319
```

```
9CE9 01        1020        DEFB 1
9CEA 5201      1030        DEFW 338
9CEC 01        1040        DEFB 1
9CED 1C01      1050        DEFW 284
9CEF 01        1060        DEFB 1
9CF0 EF00      1070        DEFW 239
9CF2 02        1080        DEFB 2
9CF3 BE00      1090        DEFW 190
9CF5 01        1100        DEFB 1
9CF6 D500      1110        DEFW 213
9CF8 01        1120        DEFB 1
9CF9 EF00      1130        DEFW 239
9CFB 01        1140        DEFB 1
9CFC 1C01      1150        DEFW 284
9CFE 06        1160        DEFB 6
9CFF D500      1170        DEFW 213
9D01 02        1180        DEFB 2
9D02 AA01      1190        DEFW 426
9D04 02        1200        DEFB 2
9D05 9201      1210        DEFW 402
9D07 01        1220        DEFB 1
9D08 7B01      1230        DEFW 379
9D0A 02        1240        DEFB 2
9D0B EF00      1250        DEFW 239
9D0D 01        1260        DEFB 1
9D0E 7B01      1270        DEFW 379
```

```
9D10 02      1280      DEFB 2
9D11 EF00    1290      DEFW 239
9D13 01      1300      DEFB 1
9D14 7B01    1310      DEFW 379
9D16 06      1320      DEFB 6
9D17 EF00    1330      DEFW 239
9D19 01      1340      DEFB 1
9D1A EF00    1350      DEFW 239
9D1C 01      1360      DEFB 1
9D1D D500    1370      DEFW 213
9D1F 01      1380      DEFB 1
9D20 C900    1390      DEFW 201
9D22 01      1400      DEFB 1
9D23 BE00    1410      DEFW 190
9D25 01      1420      DEFB 1
9D26 EF00    1430      DEFW 239
9D28 01      1440      DEFB 1
9D29 D500    1450      DEFW 213
9D2B 02      1460      DEFB 2
9D2C BE00    1470      DEFW 190
9D2E 01      1480      DEFB 1
9D2F FD00    1490      DEFW 253
9D31 02      1500      DEFB 2
9D32 D500    1510      DEFW 213
9D34 06      1520      DEFB 6
9D35 EF00    1530      DEFW 239
```

```
9D37 01        1540        DEFB 1
9D38 EF00      1550        DEFW 239
9D3A 01        1560        DEFB 1
9D3B D500      1570        DEFW 213
9D3D 01        1580        DEFB 1
9D3E BE00      1590        DEFW 190
9D40 01        1600        DEFB 1
9D41 EF00      1610        DEFW 239
9D43 01        1620        DEFB 1
9D44 D500      1630        DEFW 213
9D46 02        1640        DEFB 2
9D47 BE00      1650        DEFW 190
9D49 01        1660        DEFB 1
9D4A EF00      1670        DEFW 239
9D4C 01        1680        DEFB 1
9D4D D500      1690        DEFW 213
9D4F 01        1700        DEFB 1
9D50 EF00      1710        DEFW 239
9D52 01        1720        DEFB 1
9D53 BE00      1730        DEFW 190
9D55 01        1740        DEFB 1
9D56 EF00      1750        DEFW 239
9D58 01        1760        DEFB 1
9D59 D500      1770        DEFW 213
9D5B 02        1780        DEFB 2
9D5C BE00      1790        DEFW 190
9D5E 01        1800        DEFB 1
```

```
9D5F EF00      1810      DEFW 239
9D61 01        1820      DEFB 1
9D62 D500      1830      DEFW 213
9D64 01        1840      DEFB 1
9D65 EF00      1850      DEFW 239
9D67 01        1860      DEFB 1
9D68 BE00      1870      DEFW 190
9D6A 01        1880      DEFB 1
9D6B EF00      1890      DEFW 239
9D6D 01        1900      DEFB 1
9D6E D500      1910      DEFW 213
9D70 02        1920      DEFB 2
9D71 BE00      1930      DEFW 190
9D73 01        1940      DEFB 1
9D74 FD00      1950      DEFW 253
9D76 02        1960      DEFB 2
9D77 D500      1970      DEFW 213
9D79 06        1980      DEFB 6
9D7A EF00      1990      DEFW 239
9D7C 01        2000      DEFB 1
9D7D 7B01      2010      DEFW 379
9D7F 01        2020      DEFB 1
9D80 6601      2030      DEFW 358
9D82 01        2040      DEFB 1
9D83 5201      2050      DEFW 338
9D85 02        2060      DEFB 2
```

```
9D86 3F01      2070        DEFW 319
9D88 01        2080        DEFB 1
9D89 1C01      2090        DEFW 284
9D8B 02        2100        DEFB 2
9D8C 3F01      2110        DEFW 319
9D8E 01        2120        DEFB 1
9D8F 7B01      2130        DEFW 379
9D91 01        2140        DEFB 1
9D92 6601      2150        DEFW 358
9D94 01        2160        DEFB 1
9D95 5201      2170        DEFW 338
9D97 02        2180        DEFB 2
9D98 3F01      2190        DEFW 319
9D9A 01        2200        DEFB 1
9D9B 1C01      2210        DEFW 284
9D9D 02        2220        DEFB 2
9D9E 3F01      2230        DEFW 319
9DA0 01        2240        DEFB 1
9DA1 BE00      2250        DEFW 190
9DA3 01        2260        DEFB 1
9DA4 EF00      2270        DEFW 239
9DA6 01        2280        DEFB 1
9DA7 3F01      2290        DEFW 319
9DA9 01        2300        DEFB 1
9DAA 1C01      2310        DEFW 284
9DAC 01        2320        DEFB 1
```

```
9DAD FD00      2330        DEFW 253
9DAF 01        2340        DEFB 1
9DB0 EF00      2350        DEFW 239
9DB2 01        2360        DEFB 1
9DB3 D500      2370        DEFW 213
9DB5 01        2380        DEFB 1
9DB6 BE00      2390        DEFW 190
9DB8 01        2400        DEFB 1
9DB9 D500      2410        DEFW 213
9DBB 01        2420        DEFB 1
9DBC EF00      2430        DEFW 239
9DBE 01        2440        DEFB 1
9DBF D500      2450        DEFW 213
9DC1 05        2460        DEFB 5
9DC2 3F01      2470        DEFW 319
9DC4 01        2480        DEFB 1
9DC5 3F01      2490        DEFW 319
9DC7 01        2500        DEFB 1
9DC8 7B01      2510        DEFW 379
9DCA 01        2520        DEFB 1
9DCB 6601      2530        DEFW 358
9DCD 02        2540        DEFB 2
9DCE 3F01      2550        DEFW 319
9DD0 01        2560        DEFB 1
9DD1 1C01      2570        DEFW 284
9DD3 02        2580        DEFB 2
9DD4 3F01      2590        DEFW 319
```

```
9DD6 01        2600        DEFB 1
9DD7 7B01      2610        DEFW 379
9DD9 01        2620        DEFB 1
9DDA 6601      2630        DEFW 358
9DDC 01        2640        DEFB 1
9DDD 5201      2650        DEFW 338
9DDF 02        2660        DEFB 2
9DE0 3F01      2670        DEFW 319
9DE2 01        2680        DEFB 1
9DE3 1C01      2690        DEFW 284
9DE5 02        2700        DEFB 2
9DE6 3F01      2710        DEFW 319
9DE8 01        2720        DEFB 1
9DE9 3F01      2730        DEFW 319
9DEB 01        2740        DEFB 1
9DEC 1C01      2750        DEFW 284
9DEE 01        2760        DEFB 1
9DEF 0C01      2770        DEFW 268
9DF1 01        2780        DEFB 1
9DF2 FD00      2790        DEFW 253
9DF4 02        2800        DEFB 2
9DF5 FD00      2810        DEFW 253
9DF7 02        2820        DEFB 2
9DF8 FD00      2830        DEFW 253
9DFA 01        2840        DEFB 1
9DFB 1C01      2850        DEFW 284
9DFD 01        2860        DEFB 1
9DFE 5201      2870        DEFW 338
9E00 01        2880        DEFB 1
```

```
9E01 AA01      2890      DEFW 426
9E03 05        2900      DEFB 5
9E04 3F01      2910      DEFW 319
9E06 01        2920      DEFB 1
9E07 7B01      2930      DEFW 379
9E09 01        2940      DEFB 1
9E0A 6601      2950      DEFW 358
9E0C 01        2960      DEFB 1
9E0D 5201      2970      DEFW 338
9E0F 02        2980      DEFB 2
9E10 3F01      2990      DEFW 319
9E12 01        3000      DEFB 1
9E13 1C01      3010      DEFW 284
9E15 02        3020      DEFB 2
9E16 3F01      3030      DEFW 319
9E18 01        3040      DEFB 1
9E19 7B01      3050      DEFW 379
9E1B 01        3060      DEFB 1
9E1C 6601      3070      DEFW 358
9E1E 01        3080      DEFB 1
9E1F 5201      3090      DEFW 338
9E21 02        3100      DEFB 2
9E22 3F01      3110      DEFW 319
9E24 01        3120      DEFB 1
9E25 1C01      3130      DEFW 284
9E27 02        3140      DEFB 2
```

```
9E28 3F01      3150        DEFW 319
9E2A 01        3160        DEFB 1
9E2B BE00      3170        DEFW 190
9E2D 01        3180        DEFB 1
9E2E EF00      3190        DEFW 239
9E30 01        3200        DEFB 1
9E31 3F01      3210        DEFW 319
9E33 01        3220        DEFB 1
9E34 1C01      3230        DEFW 284
9E36 01        3240        DEFB 1
9E37 FD00      3250        DEFW 253
9E39 01        3260        DEFB 1
9E3A EF00      3270        DEFW 239
9E3C 01        3280        DEFB 1
9E3D D500      3290        DEFW 213
9E3F 01        3300        DEFB 1
9E40 BE00      3310        DEFW 190
9E42 01        3320        DEFB 1
9E43 D500      3330        DEFW 213
9E45 01        3340        DEFB 1
9E46 EF00      3350        DEFW 239
9E48 01        3360        DEFB 1
9E49 D500      3370        DEFW 213
9E4B 05        3380        DEFB 5
9E4C EF00      3390        DEFW 239
9E4E 01        3400        DEFB 1
```

```
9E4F 3F01      3410      DEFW 319
9E51 01        3420      DEFB 1
9E52 5201      3430      DEFW 338
9E54 01        3440      DEFB 1
9E55 3F01      3450      DEFW 319
9E57 02        3460      DEFB 2
9E58 EF00      3470      DEFW 239
9E5A 01        3480      DEFB 1
9E5B 1C01      3490      DEFW 284
9E5D 02        3500      DEFB 2
9E5E EF00      3510      DEFW 239
9E60 01        3520      DEFB 1
9E61 1C01      3530      DEFW 284
9E63 01        3540      DEFB 1
9E64 EF00      3550      DEFW 239
9E66 01        3560      DEFB 1
9E67 1C01      3570      DEFW 284
9E69 01        3580      DEFB 1
9E6A 3F01      3590      DEFW 319
9E6C 01        3600      DEFB 1
9E6D EF00      3610      DEFW 239
9E6F 01        3620      DEFB 1
9E70 BE00      3630      DEFW 190
9E72 02        3640      DEFB 2
9E73 9F00      3650      DEFW 159
9E75 01        3660      DEFB 1
9E76 BE00      3670      DEFW 190
```

```
9E78 01          3680          DEFB 1
9E79 EF00        3690          DEFW 239
9E7B 01          3700          DEFB 1
9E7C 3F01        3710          DEFW 319
9E7E 02          3720          DEFB 2
9E7F 1C01        3730          DEFW 284
9E81 02          3740          DEFB 2
9E82 EF00        3750          DEFW 239
9E84 01          3760          DEFB 1
9E85 BE00        3770          DEFW 190
9E87 03          3780          DEFB 3
9E88 D500        3790          DEFW 213
9E8A 06          3800          DEFB 6
9E8B EF00        3810          DEFW 239
9E8D FF          3820          DEFB 255,1
     01
                 3830          END
```

Nine – Machine Code Monitor

This program allows you to examine the contents of your Amstrad's memory. You can flick through screenfuls of ROM and RAM with it. In addition, you can alter the **contents** of memory locations, so making the program more than just a "memory viewer".

The available commands are:

'3' = Fast forwards through memory.
'E' = Slow forwards through memory.
'D' = Slow backwards through memory.
'X' = Fast backwards through memory.
'Q' = Quit. (You can't use ESC to break the program).
'ENTER' key to enter a number into an address.

There are several points to keep in mind when using the ENTER command. The address that can be altered when the ENTER key is pressed is that address at the very top of the screen display. Secondly, be careful where you attempt to alter address values, particularly between addresses 40000 and 40276, as this is where the monitor program is stored.

```
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 240
20 CLS
30 monitor=40000:scroll=40034
40 LOCATE 11,2:PRINT"Machine Code Monito
r"
```

```
50 IF PEEK(40000)=&DD AND PEEK(40001)=&6
E GOTO 90
60 FOR n=40000 TO 40276
70 READ a$:POKE n,VAL("&"+a$)
80 NEXT
90 LOCATE 11,6:INPUT"Start address";st
100 LOCATE 11,8:PRINT"For addresses &000
0-&4000"
110 LOCATE 11,9:PRINT"and &C000-&FFFF ex
amine"
120 LOCATE 11,10:PRINT"(A) ROM or (B) RA
M?"
130 rom%=0
140 IF INKEY(69)=0 THEN rom%=1:GOTO 160
150 IF INKEY(54)<>0 GOTO 140
160 start%=INT(UNT(st))-25
170 POKE scroll,25
180 CALL monitor,@rom%,@start%
190 IF rom%=99 THEN CLS:END
200 LOCATE 12,1:PRINT">  <":LOCATE 13,1
210 GOSUB 260:h$=a$
220 GOSUB 260:l$=a$
230 IF INKEY(57)*INKEY(58)*INKEY(61)=0 G
OTO 230
240 POKE start%,VAL("&"+h$+l$)
250 POKE scroll,1:GOTO 180
```

```
260 a$=UPPER$(INKEY$):IF a$<"0" OR a$>"F
" OR a$>"9" AND a$<"A" GOTO 260
270 PRINT a$;
280 RETURN
290 DATA DD,6E,00,DD,66,01,5E,23,56,DD
300 DATA 6E,02,DD,66,03,7E,B7,28,08,CD
310 DATA 00,B9,CD,06,B9,18,06,CD,03,B9
320 DATA CD,09,B9,06,19,C5,3E,19,CD,D1
330 DATA 9C,C1,10,F7,3E,39,CD,1E,BB,28
340 DATA 07,3E,19,CD,D1,9C,18,F2,3E,3F
350 DATA CD,1E,BB,28,07,3E,01,CD,D1,9C
360 DATA 18,E4,3E,3A,CD,1E,BB,28,07,3E
370 DATA 19,CD,C7,9C,18,D6,3E,3D,CD,1E
380 DATA BB,28,07,3E,01,CD,C7,9C,18,C8
390 DATA 3E,12,CD,1E,BB,20,0F,3E,43,CD
400 DATA 1E,BB,28,BA,DD,6E,02,DD,66,03
410 DATA 36,63,DD,6E,00,DD,66,01,73,23
420 DATA 72,CD,03,BB,C9,F5,01,00,50,0B
430 DATA 78,B1,20,FB,F1,F5,D5,47,CB,38
440 DATA CB,38,CB,38,CB,38,AF,CD,4D,BC
450 DATA 3E,1F,CD,5A,BB,3E,05,CD,5A,BB
460 DATA D1,F1,F5,CD,5A,BB,FE,01,28,07
470 DATA 13,21,18,00,19,18,03,1B,62,6B
480 DATA 44,CD,34,9D,45,CD,34,9D,3E,1F
490 DATA CD,5A,BB,3E,0D,CD,5A,BB,F1,F5
500 DATA CD,5A,BB,46,CD,34,9D,C1,7E,FE
```

```
510 DATA 21,F8,FE,7F,F0,3E,1F,CD,5A,BB
520 DATA 3E,13,CD,5A,BB,78,CD,5A,BB,7E
530 DATA CD,5A,BB,C9,78,E6,0F,4F,CB,38
540 DATA CB,38,CB,38,CB,38,78,06,02,FE
550 DATA 0A,F2,4C,9D,C6,30,18,02,C6,37
560 DATA CD,5A,BB,79,10,EF,C9
```

Note that the first 64 bytes of the lower ROM are identical to the first 64 bytes of RAM. This is because those first bytes constitute the low kernel jump block and are copied from ROM to RAM when the machine is switched on.

You will be asked for a starting address when the program runs. Enter this in decimal or hex. If you use hex, the address must be preceded by an '&' character. As you can see from the dump of the program in action, all addresses and values are displayed in hexadecimal.

```
Machine Code Monitor

Start address? &742

For addresses &0000-&4000
and &C000-&FFFF examine
(A) ROM or (B) RAM?

0742    53    S
0743    63    c
0744    68    h
```

```
0745    6E    n
0746    65    e
0747    69    i
0748    64    d
0749    65    e
074A    72    r
074B    00
074C    0A
074D    20
074E    41    A
074F    77    w
0750    61    a
0751    00
0752    0A
0753    20
0754    53    S
0755    6F    o
0756    6C    l
0757    61    a
0758    76    v
0759    6F    o
075A    78    x
```

```
                0001 ;          MC    MONITOR
                0002 ;
9C40            0010            ORG   40000
B900            0020 UROMON     DEFL  0B900H
B903            0030 UROMOF     DEFL  0B903H
B906            0040 LROMON     DEFL  0B906H
B909            0050 LROMOF     DEFL  0B909H
BB1E            0060 TSTKEY     DEFL  0BB1EH
BB03            0070 KRESET     DEFL  0BB03H
BC4D            0080 SCROLL     DEFL  0BC4DH
BB5A            0090 TXTOUT     DEFL  0BB5AH
9C40 DD6E00     0100            LD    L,(IX+0)
9C43 DD6601     0110            LD    H,(IX+1)
9C46 5E         0120            LD    E,(HL)
9C47 23         0130            INC   HL
9C48 56         0140            LD    D,(HL)
9C49 DD6E02     0150            LD    L,(IX+2)
9C4C DD6603     0160            LD    H,(IX+3)
9C4F 7E         0170            LD    A,(HL)
9C50 B7         0180            OR    A
9C51 2808       0190            JR    Z,ROMOFF
9C53 CD00B9     0200            CALL  UROMON
9C56 CD06B9     0210            CALL  LROMON
9C59 1806       0220            JR    ROMSET
9C5B CD03B9     0230 ROMOFF     CALL  UROMOF
9C5E CD09B9     0240            CALL  LROMOF
9C61 0619       0250 ROMSET     LD    B,25
```

```
9C63 C5       0260 SETUP  PUSH BC
9C64 3E19     0270        LD   A,25
9C66 CDD19C   0280        CALL NOWAIT
9C69 C1       0290        POP  BC
9C6A 10F7     0300        DJNZ SETUP
9C6C 3E39     0310 MAIN   LD   A,57
9C6E CD1EBB   0320        CALL TSTKEY
9C71 2807     0330        JR   Z,NOT3
9C73 3E19     0340        LD   A,25
9C75 CDD19C   0350        CALL NOWAIT
9C78 18F2     0360        JR   MAIN
9C7A 3E3F     0370 NOT3   LD   A,63
9C7C CD1EBB   0380        CALL TSTKEY
9C7F 2807     0390        JR   Z,NOTX
9C81 3E01     0400        LD   A,1
9C83 CDD19C   0410        CALL NOWAIT
9C86 18E4     0420        JR   MAIN
9C88 3E3A     0430 NOTX   LD   A,58
9C8A CD1EBB   0440        CALL TSTKEY
9C8D 2807     0450        JR   Z,NOTE
9C8F 3E19     0460        LD   A,25
9C91 CDC79C   0470        CALL PAUSE
9C94 18D6     0480        JR   MAIN
9C96 3E3D     0490 NOTE   LD   A,61
9C98 CD1EBB   0500        CALL TSTKEY
9C9B 2807     0510        JR   Z,NOTD
9C9D 3E01     0520        LD   A,1
```

```
9C9F CDC79C   0530         CALL PAUSE
9CA2 18C8     0540         JR   MAIN
9CA4 3E12     0550 NOTD    LD   A,18
9CA6 CD1EBB   0560         CALL TSTKEY
9CA9 200F     0570         JR   NZ,ENTER
9CAB 3E43     0580         LD   A,67
9CAD CD1EBB   0590         CALL TSTKEY
9CB0 28BA     0600         JR   Z,MAIN
9CB2 DD6E02   0610         LD   L,(IX+2)
9CB5 DD6603   0620         LD   H,(IX+3)
9CB8 3663     0630         LD   (HL),99
9CBA DD6E00   0640 ENTER   LD   L,(IX+0)
9CBD DD6601   0650         LD   H,(IX+1)
9CC0 73       0660         LD   (HL),E
9CC1 23       0670         INC  HL
9CC2 72       0680         LD   (HL),D
9CC3 CD03BB   0690         CALL KRESET
9CC6 C9       0700         RET
9CC7 F5       0710 PAUSE   PUSH AF
9CC8 010050   0720         LD   BC,5000H
9CCB 0B       0730 DELAY   DEC  BC
9CCC 78       0740         LD   A,B
9CCD B1       0750         OR   C
9CCE 20FB     0760         JR   NZ,DELAY
9CD0 F1       0770         POP  AF
9CD1 F5       0780 NOWAIT  PUSH AF
9CD2 D5       0790         PUSH DE
```

```
9CD3 47       0800        LD   B,A
9CD4 CB38     0810        SRL  B
9CD6 CB38     0820        SRL  B
9CD8 CB38     0830        SRL  B
9CDA CB38     0840        SRL  B
9CDC AF       0850        XOR  A
9CDD CD4DBC   0860        CALL SCROLL
9CE0 3E1F     0870        LD   A,31
9CE2 CD5ABB   0880        CALL TXTOUT
9CE5 3E05     0890        LD   A,5
9CE7 CD5ABB   0900        CALL TXTOUT
9CEA D1       0910        POP  DE
9CEB F1       0920        POP  AF
9CEC F5       0930        PUSH AF
9CED CD5ABB   0940        CALL TXTOUT
9CF0 FE01     0950        CP   1
9CF2 2807     0960        JR   Z,DOWN
9CF4 13       0970        INC  DE
9CF5 211800   0980        LD   HL,24
9CF8 19       0990        ADD  HL,DE
9CF9 1803     1000        JR   PRINT
9CFB 1B       1010 DOWN   DEC  DE
9CFC 62       1020        LD   H,D
9CFD 6B       1030        LD   L,E
9CFE 44       1040 PRINT  LD   B,H
9CFF CD349D   1050        CALL HEXOUT
9D02 45       1060        LD   B,L
```

```
9D03 CD349D    1070          CALL HEXOUT
9D06 3E1F      1080          LD   A,31
9D08 CD5ABB    1090          CALL TXTOUT
9D0B 3E0D      1100          LD   A,13
9D0D CD5ABB    1110          CALL TXTOUT
9D10 F1        1120          POP  AF
9D11 F5        1130          PUSH AF
9D12 CD5ABB    1140          CALL TXTOUT
9D15 46        1150          LD   B,(HL)
9D16 CD349D    1160          CALL HEXOUT
9D19 C1        1170          POP  BC
9D1A 7E        1180          LD   A,(HL)
9D1B FE21      1190          CP   33
9D1D F8        1200          RET  M
9D1E FE7F      1210          CP   127
9D20 F0        1220          RET  P
9D21 3E1F      1230          LD   A,31
9D23 CD5ABB    1240          CALL TXTOUT
9D26 3E13      1250          LD   A,19
9D28 CD5ABB    1260          CALL TXTOUT
9D2B 78        1270          LD   A,B
9D2C CD5ABB    1280          CALL TXTOUT
9D2F 7E        1290          LD   A,(HL)
9D30 CD5ABB    1300          CALL TXTOUT
9D33 C9        1310          RET
9D34 78        1320 HEXOUT   LD   A,B
9D35 E60F      1330          AND  15
```

```
9D37 4F       1340         LD   C,A
9D38 CB38     1350         SRL  B
9D3A CB38     1360         SRL  B
9D3C CB38     1370         SRL  B
9D3E CB38     1380         SRL  B
9D40 78       1390         LD   A,B
9D41 0602     1400         LD   B,2
9D43 FE0A     1410 DIGIT   CP   10
9D45 F24C9D   1420         JP   P,LETTER
9D48 C630     1430         ADD  A,48
9D4A 1802     1440         JR   PRNTCH
9D4C C637     1450 LETTER  ADD  A,55
9D4E CD5ABB   1460 PRNTCH  CALL TXTOUT
9D51 79       1470         LD   A,C
9D52 10EF     1480         DJNZ DIGIT
9D54 C9       1490         RET
              1500         END
```

Ten – Box Scrolls

We are quite proud of these routines. You should find them very handy for any programs which require moving graphics.

Scroll Box Up

Every machine code book worth its salt includes some sort of scroll routine. This is usually restricted to scrolling a row, a column or the whole screen. Our routine for **Scroll Box Up** (and the three other direction routines, down, left and right) allows you to specify a "box", or window on-screen, and make that box scroll smoothly.

A smooth scroll is a pixel by pixel scroll. Even in machine code it is not dramatically fast, but should be rapid enough for most requirements. Its great asset is the smoothness of the scroll, compared with a character by character scroll.

The box is defined in a similar way to the WINDOW command. The format is:

CALL 40000, L, R, U, D

In this, L stands for the leftmost co-ordinate of the box, R for the rightmost co-ordinate, U for the top, and D for the bottom co-ordinate. Therefore, the box can only be square or rectangular. Obviously, L must be equal to, or smaller than R; and U must be equal to, or smaller than D. Note that L, R, U and D can be either numeric variables or numbers.

```
1 '              SCROLL BOX UP
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 240
20 FOR n=40000 TO 40145
30 READ a$:POKE n,VAL("&"+a$)
40 NEXT
50 DATA DD,6E,02,DD,66,06,2D,25,DD,7E
60 DATA 04,94,4F,DD,7E,00,95,57,CD,1A
70 DATA BC,3E,00,81,10,FD,5F,18,38,D5
80 DATA 43,5D,7C,C6,38,57,7D,C6,50,6F
90 DATA 30,0A,24,7C,E6,07,20,04,7C,D6
100 DATA 08,67,E5,7E,12,1C,20,0A,14,7A
110 DATA E6,07,20,04,7A,D6,08,57,2C,20
120 DATA 0A,24,7C,E6,07,20,04,7C,D6,08
130 DATA 67,10,E2,E1,D1,D5,E5,4B,E5,06
140 DATA 07,5D,54,7C,C6,08,67,7E,12,10
150 DATA F6,E1,2C,20,0A,24,7C,E6,07,20
160 DATA 04,7C,D6,08,67,0D,20,E2,E1,D1
170 DATA 15,20,A2,7C,C6,38,67,43,36,00
180 DATA 2C,20,0A,24,7C,E6,07,20,04,7C
190 DATA D6,08,67,10,EF,C9
```

When you make the CALL, the specified box would scroll up one single pixel with the top line disappearing and an extra blank line appearing at the bottom of the box. One single scroll is not enough for most uses. To scroll a number of times, we use a FOR/NEXT loop.

There are few limitations in using this program. It works in any

screen mode, and several boxes can be scrolled at the same time. Of course, the more boxes you have on-screen, as well as the larger the boxes you have, the slower the scroll procedure works. Therefore, it is best to restrict the size and number of boxes to that which is really necessary. For a weird effect, try scrolling two boxes that overlap each other.

```
                0010 ;SCRL   BOX   UP
                0020 ;
9C40            0030         ORG   40000
BC1A            0040 CHRPOS  DEFL  0BC1AH
9C40 DD6E02     0050         LD    L,(IX+2)
9C43 DD6606     0060         LD    H,(IX+6)
9C46 2D         0070         DEC   L
9C47 25         0080         DEC   H
9C48 DD7E04     0090         LD    A,(IX+4)
9C4B 9d         0100         SUB   H
9C4C 4F         0110         LD    C,A
9C4D DD7E00     0120         LD    A,(IX+0)
9C50 95         0130         SUB   L
9C51 57         0140         LD    D,A
9C52 CD1ABC     0150         CALL  CHRPOS
9C55 3E00       0160         LD    A,0
9C57 81         0170 CHRWID  ADD   A,C
9C58 10FD       0180         DJNZ  CHRWID
9C5A 5F         0190         LD    E,A
9C5B 1838       0200         JR    START
9C5D D5         0210 NXTROW  PUSH  DE
9C5E 43         0220         LD    B,E
```

```
9C5F 5D      0230         LD   E,L
9C60 7C      0240         LD   A,H
9C61 C638    0250         ADD  A,56
9C63 57      0260         LD   D,A
9C64 7D      0270         LD   A,L
9C65 C650    0280         ADD  A,80
9C67 6F      0290         LD   L,A
9C68 300A    0300         JR   NC,NEWLIN
9C6A 24      0310         INC  H
9C6B 7C      0320         LD   A,H
9C6C E607    0330         AND  7
9C6E 2004    0340         JR   NZ,NEWLIN
9C70 7C      0350         LD   A,H
9C71 D608    0360         SUB  8
9C73 67      0370         LD   H,A
9C74 E5      0380 NEWLIN  PUSH HL
9C75 7E      0390 LASTLN  LD   A,(HL)
9C76 12      0400         LD   (DE),A
9C77 1C      0410         INC  E
9C78 200A    0420         JR   NZ,DEOK
9C7A 14      0430         INC  D
9C7B 7A      0440         LD   A,D
9C7C E607    0450         AND  7
9C7E 2004    0460         JR   NZ,DEOK
9C80 7A      0470         LD   A,D
9C81 D608    0480         SUB  8
```

```
9C83 57      0490        LD   D,A
9C84 2C      0500 DEOK   INC  L
9C85 200A    0510        JR   NZ,HLOK
9C87 24      0520        INC  H
9C88 7C      0530        LD   A,H
9C89 E607    0540        AND  7
9C8B 2004    0550        JR   NZ,HLOK
9C8D 7C      0560        LD   A,H
9C8E D608    0570        SUB  8
9C90 67      0580        LD   H,A
9C91 10E2    0590 HLOK   DJNZ LASTLN
9C93 E1      0600        POP  HL
9C94 D1      0610        POP  DE
9C95 D5      0620 START  PUSH DE
9C96 E5      0630        PUSH HL
9C97 4B      0640        LD   C,E
9C98 E5      0650 NXTCHR PUSH HL
9C99 0607    0660        LD   B,7
9C9B 5D      0670 NXTLIN LD   E,L
9C9C 54      0680        LD   D,H
9C9D 7C      0690        LD   A,H
9C9E C608    0700        ADD  A,8
9CA0 67      0710        LD   H,A
9CA1 7E      0720        LD   A,(HL)
9CA2 12      0730        LD   (DE),A
9CA3 10F6    0740        DJNZ NXTLIN
```

```
9CA5 E1      0750          POP  HL
9CA6 2C      0760          INC  L
9CA7 200A    0770          JR   NZ,OK
9CA9 24      0780          INC  H
9CAA 7C      0790          LD   A,H
9CAB E607    0800          AND  7
9CAD 2004    0810          JR   NZ,OK
9CAF 7C      0820          LD   A,H
9CB0 D608    0830          SUB  8
9CB2 67      0840          LD   H,A
9CB3 0D      0850 OK       DEC  C
9CB4 20E2    0860          JR   NZ,NXTCHR
9CB6 E1      0870          POP  HL
9CB7 D1      0880          POP  DE
9CB8 15      0890          DEC  D
9CB9 20A2    0900          JR   NZ,NXTROW
9CBB 7C      0910          LD   A,H
9CBC C638    0920          ADD  A,56
9CBE 67      0930          LD   H,A
9CBF 43      0940          LD   B,E
9CC0 3600    0950 BLANK    LD   (HL),0
9CC2 2C      0960          INC  L
9CC3 200A    0970          JR   NZ,CONT
9CC5 24      0980          INC  H
9CC6 7C      0990          LD   A,H
9CC7 E607    1000          AND  7
9CC9 2004    1010          JR   NZ,CONT
```

```
9CCB 7C        1020          LD    A,H
9CCC D608      1030          SUB   8
9CCE 67        1040          LD    H,A
9CCF 10EF      1050 CONT     DJNZ  BLANK
9CD1 C9        1060          RET
               1070          END
```

Scroll Box Down

This routine is similar in design and use to the previous routine used for scrolling a box up the screen. It works in all three screen modes as does Scroll Box Up and the format to use it is exactly the same:

CALL **40000**, L R, U, D

```
1 '          SCROLL BOX DOWN
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 240
20 FOR n=40000 TO 40151
30 READ a$:POKE n,VAL("&"+a$)
40 NEXT
50 DATA DD,6E,00,DD,66,06,2D,25,DD,7E
60 DATA 04,94,4F,7D,C6,02,DD,96,02,57
70 DATA CD,1A,BC,7C,C6,38,67,3E,00,81
80 DATA 10,FD,5F,18,38,D5,43,5D,7C,D6
90 DATA 38,57,7D,D6,50,6F,30,0A,7C,25
100 DATA E6,07,20,04,7C,C6,08,67,E5,7E
110 DATA 12,1C,20,0A,14,7A,E6,07,20,04
```

```
120 DATA 7A,D6,08,57,2C,20,0A,24,7C,E6
130 DATA 07,20,04,7C,D6,08,67,10,E2,E1
140 DATA D1,D5,E5,4B,E5,06,07,5D,54,7C
150 DATA D6,08,67,7E,12,10,F6,E1,2C,20
160 DATA 0A,24,7C,E6,07,20,04,7C,D6,08
170 DATA 67,0D,20,E2,E1,D1,15,20,A2,7C
180 DATA D6,38,67,43,36,00,2C,20,0A,24
190 DATA 7C,E6,07,20,04,7C,D6,08,67,10
200 DATA EF,C9
```

Once again, L is left, R is right, U is up and D is down. As with the previous routine, a single CALL will result in a one pixel-line scroll so a loop will be necessary if you intend to do a lot of scrolling.

```
                    0001 ;        SCRL DOWN
                    0002 ;
9C40                0010          ORG  40000
BC1A                0020 CHRPOS   DEFL 0BC1AH
9C40 DD6E00         0030          LD   L,(IX+0)
9C43 DD6606         0040          LD   H,(IX+6)
9C46 2D             0050          DEC  L
9C47 25             0060          DEC  H
9C48 DD7E04         0070          LD   A,(IX+4)
9C4B 94             0080          SUB  H
9C4C 4F             0090          LD   C,A
9C4D 7D             0100          LD   A,L
9C4E C602           0110          ADD  A,2
9C50 DD9602         0120          SUB  (IX+2)
```

```
9C53 57       0130         LD   D,A
9C54 CD1ABC   0140         CALL CHRPOS
9C57 7C       0150         LD   A,H
9C58 C638     0160         ADD  A,56
9C5A 67       0170         LD   H,A
9C5B 3E00     0180         LD   A,0
9C5D 81       0190 CHRWID  ADD  A,C
9C5E 10FD     0200         DJNZ CHRWID
9C60 5F       0210         LD   E,A
9C61 1838     0220         JR   START
9C63 D5       0230 NXTROW  PUSH DE
9C64 43       0240         LD   B,E
9C65 5D       0250         LD   E,L
9C66 7C       0260         LD   A,H
9C67 D638     0270         SUB  56
9C69 57       0280         LD   D,A
9C6A 7D       0290         LD   A,L
9C6B D650     0300         SUB  80
9C6D 6F       0310         LD   L,A
9C6E 300A     0320         JR   NC,NEWLIN
9C70 7C       0330         LD   A,H
9C71 25       0340         DEC  H
9C72 E607     0350         AND  7
9C74 2004     0360         JR   NZ,NEWLIN
9C76 7C       0370         LD   A,H
9C77 C608     0380         ADD  A,8
9C79 67       0390         LD   H,A
```

```
9C7A E5       0400 NEWLIN PUSH HL
9C7B 7E       0410 LASTLN LD   A,(HL)
9C7C 12       0420        LD   (DE),A
9C7D 1C       0430        INC  E
9C7E 200A     0440        JR   NZ,DEOK
9C80 14       0450        INC  D
9C81 7A       0460        LD   A,D
9C82 E607     0470        AND  7
9C84 2004     0480        JR   NZ,DEOK
9C86 7A       0490        LD   A,D
9C87 D608     0500        SUB  8
9C89 57       0510        LD   D,A
9C8A 2C       0520 DEOK   INC  L
9C8B 200A     0530        JR   NZ,HLOK
9C8D 24       0540        INC  H
9C8E 7C       0550        LD   A,H
9C8F E607     0560        AND  7
9C91 2004     0570        JR   NZ,HLOK
9C93 7C       0580        LD   A,H
9C94 D608     0590        SUB  8
9C96 67       0600        LD   H,A
9C97 10E2     0610 HLOK   DJNZ LASTLN
9C99 E1       0620        POP  HL
9C9A D1       0630        POP  DE
9C9B D5       0640 START  PUSH DE
9C9C E5       0650        PUSH HL
9C9D 4B       0660        LD   C,E
```

```
9C9E E5       0670 NXTCHR PUSH HL
9C9F 0607     0680        LD   B,7
9CA1 5D       0690 NXTLIN LD   E,L
9CA2 54       0700        LD   D,H
9CA3 7C       0710        LD   A,H
9CA4 D608     0720        SUB  8
9CA6 67       0730        LD   H,A
9CA7 7E       0740        LD   A,(HL)
9CA8 12       0750        LD   (DE),A
9CA9 10F6     0760        DJNZ NXTLIN
9CAB E1       0770        POP  HL
9CAC 2C       0780        INC  L
9CAD 200A     0790        JR   NZ,OK
9CAF 24       0800        INC  H
9CB0 7C       0810        LD   A,H
9CB1 E607     0820        AND  7
9CB3 2004     0830        JR   NZ,OK
9CB5 7C       0840        LD   A,H
9CB6 D608     0850        SUB  8
9CB8 67       0860        LD   H,A
9CB9 0D       0870 OK     DEC  C
9CBA 20E2     0880        JR   NZ,NXTCHR
9CBC E1       0890        POP  HL
9CBD D1       0900        POP  DE
9CBE 15       0910        DEC  D
9CBF 20A2     0920        JR   NZ,NXTROW
```

```
9CC1 7C      0930       LD   A,H
9CC2 D638    0940       SUB  56
9CC4 67      0950       LD   H,A
9CC5 43      0960       LD   B,E
9CC6 3600    0970 BLANK LD   (HL),0
9CC8 2C      0980       INC  L
9CC9 200A    0990       JR   NZ,CONT
9CCB 24      1000       INC  H
9CCC 7C      1010       LD   A,H
9CCD E607    1020       AND  7
9CCF 2004    1030       JR   NZ,CONT
9CD1 7C      1040       LD   A,H
9CD2 D608    1050       SUB  8
9CD4 67      1060       LD   H,A
9CD5 10EF    1070 CONT  DJNZ BLANK
9CD7 C9      1080       RET
             1090       END
```

Scroll Box Left

Though scroll left (and right) work in exactly the same way as scroll up and down, the **routine** to create the effect has some differences. This is mainly due to the horizontal scrolling movement as opposed to the vertical movement of the previous two routines.

The format to use it is exactly the same:

CALL 40000, L, R, U, D

```
1 '          SCROLL BOX LEFT
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 240
20 FOR n=40000 TO 40271
30 READ a$:POKE n,VAL("&"+a$)
40 NEXT
50 DATA DD,6E,02,DD,66,06,2D,25,DD,7E
60 DATA 04,94,4F,DD,7E,00,95,57,CD,1A
70 DATA BC,3E,00,81,10,FD,47,CD,11,BC
80 DATA DA,FA,9C,28,48,DD,6E,02,DD,66
90 DATA 04,2D,25,C5,CD,1A,BC,C1,0E,08
100 DATA C5,E5,B7,CB,16,F5,7D,2D,B7,20
110 DATA 0A,7C,25,E6,07,20,04,7C,C6,08
120 DATA 67,F1,10,EB,E1,7C,C6,08,67,C1
130 DATA 0D,20,DF,7C,D6,40,67,7D,C6,50
140 DATA 6F,30,0A,24,7C,E6,07,20,04,7C
150 DATA D6,08,67,15,20,C6,C9,05,D5,0E
160 DATA 08,C5,E5,CB,26,5D,54,2C,20,0A
170 DATA 24,7C,E6,07,20,04,7C,D6,08,67
180 DATA 1A,CB,26,38,04,CB,A7,18,02,CB
190 DATA E7,CB,66,28,02,CB,C7,12,10,DD
200 DATA CB,A6,E1,7C,C6,08,67,C1,0D,20
210 DATA CE,7C,D6,40,67,7D,C6,50,6F,30
220 DATA 0A,24,7C,E6,07,20,04,7C,D6,08
230 DATA 67,D1,15,20,B3,C9,05,D5,0E,08
240 DATA C5,E5,C5,5D,54,2C,20,0A,24,7C
250 DATA E6,07,20,04,7C,D6,08,67,7E,1F
```

```
260 DATA 4F,1A,17,17,06,04,17,CB,21,CB
270 DATA 21,17,10,F8,12,C1,10,DC,7E,17
280 DATA 06,04,07,CB,27,10,FB,77,E1,7C
290 DATA C6,08,67,C1,0D,20,C7,7C,D6,40
300 DATA 67,7D,C6,50,6F,30,0A,24,7C,E6
310 DATA 07,20,04,7C,D6,08,67,D1,15,20
320 DATA AC,C9
```

As usual, L is left, R is right, U is up and D is down. As with the other routines, a single CALL will result in a one pixel-line scroll so a loop will be necessary if you intend to do a lot of scrolling. This scroll also works in all three screen modes as does **Scroll Right.**

```
                    0001 ;        SCRL LEFT
                    0002 ;
9C40                0010          ORG  40000
BC1A                0020 CHRPOS   DEFL 0BC1AH
BC11                0030 SCMODE   DEFL 0BC11H
9C40 DD6E02         0040          LD   L,(IX+2)
9C43 DD6606         0050          LD   H,(IX+6)
9C46 2D             0060          DEC  L
9C47 25             0070          DEC  H
9C48 DD7E04         0080          LD   A,(IX+4)
9C4B 94             0090          SUB  H
9C4C 4F             0100          LD   C,A
9C4D DD7E00         0110          LD   A,(IX+0)
9C50 95             0120          SUB  L
9C51 57             0130          LD   D,A
```

```
9C52 CD1ABC    0140          CALL  CHRPOS
9C55 3E00      0150          LD    A,0
9C57 81        0160 CHRWID   ADD   A,C
9C58 10FD      0170          DJNZ  CHRWID
9C5A 47        0180          LD    B,A
9C5B CD11BC    0190          CALL  SCMODE
9C5E DAFA9C    0200          JP    C,MODE0
9C61 2848      0210          JR    Z,MODE1
9C63 DD6E02    0220          LD    L,(IX+2)
9C66 DD6604    0230          LD    H,(IX+4)
9C69 2D        0240          DEC   L
9C6A 25        0250          DEC   H
9C6B C5        0260          PUSH  BC
9C6C CD1ABC    0270          CALL  CHRPOS
9C6F C1        0280          POP   BC
9C70 0E08      0290 ROW2     LD    C,8
9C72 C5        0300 LINE2    PUSH  BC
9C73 E5        0310          PUSH  HL
9C74 B7        0320          OR    A
9C75 CB16      0330 BYTE2    RL    (HL)
9C77 F5        0340          PUSH  AF
9C78 7D        0350          LD    A,L
9C79 2D        0360          DEC   L
9C7A B7        0370          OR    A
9C7B 200A      0380          JR    NZ,OK2
9C7D 7C        0390          LD    A,H
9C7E 25        0400          DEC   H
```

```
9C7F E607      0410          AND  7
9C81 2004      0420          JR   NZ,OK2
9C83 7C        0430          LD   A,H
9C84 C608      0440          ADD  A,8
9C86 67        0450          LD   H,A
9C87 F1        0460 OK2      POP  AF
9C88 10EB      0470          DJNZ BYTE2
9C8A E1        0480          POP  HL
9C8B 7C        0490          LD   A,H
9C8C C608      0500          ADD  A,8
9C8E 67        0510          LD   H,A
9C8F C1        0520          POP  BC
9C90 0D        0530          DEC  C
9C91 20DF      0540          JR   NZ,LINE2
9C93 7C        0550          LD   A,H
9C94 D640      0560          SUB  64
9C96 67        0570          LD   H,A
9C97 7D        0580          LD   A,L
9C98 C650      0590          ADD  A,80
9C9A 6F        0600          LD   L,A
9C9B 300A      0610          JR   NC,DONE2
9C9D 24        0620          INC  H
9C9E 7C        0630          LD   A,H
9C9F E607      0640          AND  7
9CA1 2004      0650          JR   NZ,DONE2
9CA3 7C        0660          LD   A,H
9CA4 D608      0670          SUB  8
```

```
9CA6 67       0680         LD   H,A
9CA7 15       0690 DONE2   DEC  D
9CA8 20C6     0700         JR   NZ,ROW2
9CAA C9       0710         RET
9CAB 05       0720 MODE1   DEC  B
9CAC D5       0730 ROW1    PUSH DE
9CAD 0E08     0740         LD   C,8
9CAF C5       0750 LINE1   PUSH BC
9CB0 E5       0760         PUSH HL
9CB1 CB26     0770         SLA  (HL)
9CB3 5D       0780 BYTE1   LD   E,L
9CB4 54       0790         LD   D,H
9CB5 2C       0800         INC  L
9CB6 200A     0810         JR   NZ,OK1
9CB8 24       0820         INC  H
9CB9 7C       0830         LD   A,H
9CBA E607     0840         AND  7
9CBC 2004     0850         JR   NZ,OK1
9CBE 7C       0860         LD   A,H
9CBF D608     0870         SUB  8
9CC1 67       0880         LD   H,A
9CC2 1A       0890 OK1     LD   A,(DE)
9CC3 CB26     0900         SLA  (HL)
9CC5 3804     0910         JR   C,SETBIT
9CC7 CBA7     0920         RES  4,A
9CC9 1802     0930         JR   BITOK
9CCB CBE7     0940 SETBIT  SET  4,A
```

```
9CCD CB66     0950 BITOK   BIT   4,(HL)
9CCF 2802     0960         JR    Z,BITSET
9CD1 CBC7     0970         SET   0,A
9CD3 12       0980 BITSET  LD    (DE),A
9CD4 10DD     0990         DJNZ  BYTE1
9CD6 CBA6     1000         RES   4,(HL)
9CD8 E1       1010         POP   HL
9CD9 7C       1020         LD    A,H
9CDA C608     1030         ADD   A,8
9CDC 67       1040         LD    H,A
9CDD C1       1050         POP   BC
9CDE 0D       1060         DEC   C
9CDF 20CE     1070         JR    NZ,LINE1
9CE1 7C       1080         LD    A,H
9CE2 D640     1090         SUB   64
9CE4 67       1100         LD    H,A
9CE5 7D       1110         LD    A,L
9CE6 C650     1120         ADD   A,80
9CE8 6F       1130         LD    L,A
9CE9 300A     1140         JR    NC,DONE1
9CEB 24       1150         INC   H
9CEC 7C       1160         LD    A,H
9CED E607     1170         AND   7
9CEF 2004     1180         JR    NZ,DONE1
9CF1 7C       1190         LD    A,H
9CF2 D608     1200         SUB   8
```

```
9CF4 67      1210       LD   H,A
9CF5 D1      1220 DONE1 POP  DE
9CF6 15      1230       DEC  D
9CF7 20B3    1240       JR   NZ,ROW1
9CF9 C9      1250       RET
9CFA 05      1260 MODE0 DEC  B
9CFB D5      1270 ROW0  PUSH DE
9CFC 0E08    1280       LD   C,8
9CFE C5      1290 LINE0 PUSH BC
9CFF E5      1300       PUSH HL
9D00 C5      1310 BYTE0 PUSH BC
9D01 5D      1320       LD   E,L
9D02 54      1330       LD   D,H
9D03 2C      1340       INC  L
9D04 200A    1350       JR   NZ,OK0
9D06 24      1360       INC  H
9D07 7C      1370       LD   A,H
9D08 E607    1380       AND  7
9D0A 2004    1390       JR   NZ,OK0
9D0C 7C      1400       LD   A,H
9D0D D608    1410       SUB  8
9D0F 67      1420       LD   H,A
9D10 7E      1430 OK0   LD   A,(HL)
9D11 1F      1440       RRA
9D12 4F      1450       LD   C,A
9D13 1A      1460       LD   A,(DE)
9D14 17      1470       RLA
```

```
9D15 17      1480         RLA
9D16 0604    1490         LD   B,4
9D18 17      1500 NXTROT  RLA
9D19 CB21    1510         SLA  C
9D1B CB21    1520         SLA  C
9D1D 17      1530         RLA
9D1E 10F8    1540         DJNZ NXTROT
9D20 12      1550         LD   (DE),A
9D21 C1      1560         POP  BC
9D22 10DC    1570         DJNZ BYTE0
9D24 7E      1580         LD   A,(HL)
9D25 17      1590         RLA
9D26 0604    1600         LD   B,4
9D28 07      1610 LSTBYT  RLCA
9D29 CB27    1620         SLA  A
9D2B 10FB    1630         DJNZ LSTBYT
9D2D 77      1640         LD   (HL),A
9D2E E1      1650         POP  HL
9D2F 7C      1660         LD   A,H
9D30 C608    1670         ADD  A,8
9D32 67      1680         LD   H,A
9D33 C1      1690         POP  BC
9D34 0D      1700         DEC  C
9D35 20C7    1710         JR   NZ,LINE0
9D37 7C      1720         LD   A,H
9D38 D640    1730         SUB  64
9D3A 67      1740         LD   H,A
```

```
9D3B 7D        1750          LD    A,L
9D3C C650      1760          ADD   A,80
9D3E 6F        1770          LD    L,A
9D3F 300A      1780          JR    NC,DONE0
9D41 24        1790          INC   H
9D42 7C        1800          LD    A,H
9D43 E607      1810          AND   7
9D45 2004      1820          JR    NZ,DONE0
9D47 7C        1830          LD    A,H
9D48 D608      1840          SUB   8
9D4A 67        1850          LD    H,A
9D4B D1        1860 DONE0    POP   DE
9D4C 15        1870          DEC   D
9D4D 20AC      1880          JR    NZ,ROW0
9D4F C9        1890          RET
               1900          END
```

Scroll Box Right

With scroll up, scroll down and scroll left, it's not difficult to work out what this routine will do.

Once again, the format to use it is exactly the same as the other routines:

CALL 40000, L, R, U, D

As with the other routines, a single CALL will result in a one pixel-line scroll so a loop will be necessary if you intend to do a lot of scrolling.

```
1 '           SCROLL BOX RIGHT
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 240
20 FOR n=40000 TO 40277
30 READ a$:POKE n,VAL("&"+a$)
40 NEXT
50 DATA DD,6E,02,DD,66,04,2D,25,7C,C6
60 DATA 02,DD,96,06,4F,DD,7E,00,95,57
70 DATA CD,1A,BC,2B,3E,00,81,23,10,FC
80 DATA 47,CD,11,BC,DA,FE,9C,28,46,DD
90 DATA 6E,02,DD,66,06,2D,25,C5,CD,1A
100 DATA BC,C1,0E,08,C5,E5,B7,CB,1E,F5
110 DATA 2C,20,0A,24,7C,E6,07,20,04,7C
120 DATA D6,08,67,F1,10,ED,E1,7C,C6,08
130 DATA 67,C1,0D,20,E1,7C,D6,40,67,7D
140 DATA C6,50,6F,30,0A,24,7C,E6,07,20
150 DATA 04,7C,D6,08,67,15,20,C8,C9,05
160 DATA D5,0E,08,C5,E5,CB,3E,5D,54,7D
170 DATA 2D,B7,20,0A,7C,25,E6,07,20,04
180 DATA 7C,C6,08,67,1A,CB,3E,38,04,CB
190 DATA 9F,18,02,CB,DF,CB,5E,28,02,CB
200 DATA FF,12,10,DB,CB,9E,E1,7C,C6,08
210 DATA 67,C1,0D,20,CC,7C,D6,40,67,7D
220 DATA C6,50,6F,30,0A,24,7C,E6,07,20
230 DATA 04,7C,D6,08,67,D1,15,20,B1,C9
240 DATA 05,D5,0E,08,C5,E5,C5,5D,54,7D
250 DATA 2D,B7,20,0A,7C,25,E6,07,20,04
```

```
260 DATA 7C,C6,08,67,7E,17,4F,1A,1F,1F
270 DATA 06,04,1F,CB,39,CB,39,1F,10,F8
280 DATA 12,C1,10,DA,7E,1F,06,04,0F,CB
290 DATA 3F,10,FB,77,E1,7C,C6,08,67,C1
300 DATA 0D,20,C5,7C,D6,40,67,7D,C6,50
310 DATA 6F,30,0A,24,7C,E6,07,20,04,7C
320 DATA D6,08,67,D1,15,20,AA,C9
```

```
                    0001 ;        SCRL RIGHT
                    0002 ;
9C40                0010          ORG  40000
BC1A                0020 CHRPOS   DEFL 0BC1AH
BC11                0030 SCMODE   DEFL 0BC11H
9C40 DD6E02         0040          LD   L,(IX+2)
9C43 DD6604         0050          LD   H,(IX+4)
9C46 2D             0060          DEC  L
9C47 25             0070          DEC  H
9C48 7C             0080          LD   A,H
9C49 C602           0090          ADD  A,2
9C4B DD9606         0100          SUB  (IX+6)
9C4E 4F             0110          LD   C,A
9C4F DD7E00         0120          LD   A,(IX+0)
9C52 95             0130          SUB  L
9C53 57             0140          LD   D,A
```

```
9C54 CD1ABC    0150         CALL  CHRPOS
9C57 2B        0160         DEC   HL
9C58 3E00      0170         LD    A,0
9C5A 81        0180 CHRWID  ADD   A,C
9C5B 23        0190         INC   HL
9C5C 10FC      0200         DJNZ  CHRWID
9C5E 47        0210         LD    B,A
9C5F CD11BC    0220         CALL  SCMODE
9C62 DAFE9C    0230         JP    C,MODE0
9C65 2846      0240         JR    Z,MODE1
9C67 DD6E02    0250         LD    L,(IX+2)
9C6A DD6606    0260         LD    H,(IX+6)
9C6D 2D        0270         DEC   L
9C6E 25        0280         DEC   H
9C6F C5        0290         PUSH  BC
9C70 CD1ABC    0300         CALL  CHRPOS
9C73 C1        0310         POP   BC
9C74 0E08      0320 ROW2    LD    C,8
9C76 C5        0330 LINE2   PUSH  BC
9C77 E5        0340         PUSH  HL
9C78 B7        0350         OR    A
9C79 CB1E      0360 BYTE2   RR    (HL)
9C7B F5        0370         PUSH  AF
9C7C 2C        0380         INC   L
9C7D 200A      0390         JR    NZ,OK2
9C7F 24        0400         INC   H
```

```
9C80 7C        0410          LD    A,H
9C81 E607      0420          AND   7
9C83 2004      0430          JR    NZ,OK2
9C85 7C        0440          LD    A,H
9C86 D608      0450          SUB   8
9C88 67        0460          LD    H,A
9C89 F1        0470 OK2      POP   AF
9C8A 10ED      0480          DJNZ  BYTE2
9C8C E1        0490          POP   HL
9C8D 7C        0500          LD    A,H
9C8E C608      0510          ADD   A,8
9C90 67        0520          LD    H,A
9C91 C1        0530          POP   BC
9C92 0D        0540          DEC   C
9C93 20E1      0550          JR    NZ,LINE2
9C95 7C        0560          LD    A,H
9C96 D640      0570          SUB   64
9C98 67        0580          LD    H,A
9C99 7D        0590          LD    A,L
9C9A C650      0600          ADD   A,80
9C9C 6F        0610          LD    L,A
9C9D 300A      0620          JR    NC,DONE2
9C9F 24        0630          INC   H
9CA0 7C        0640          LD    A,H
9CA1 E607      0650          AND   7
9CA3 2004      0660          JR    NZ,DONE2
9CA5 7C        0670          LD    A,H
```

```
9CA6 D608      0680          SUB   8
9CA8 67        0690          LD    H,A
9CA9 15        0700 DONE2    DEC   D
9CAA 20C8      0710          JR    NZ,ROW2
9CAC C9        0720          RET
9CAD 05        0730 MODE1    DEC   B
9CAE D5        0740 ROW1     PUSH  DE
9CAF 0E08      0750          LD    C,8
9CB1 C5        0760 LINE1    PUSH  BC
9CB2 E5        0770          PUSH  HL
9CB3 CB3E      0780          SRL   (HL)
9CB5 5D        0790 BYTE1    LD    E,L
9CB6 54        0800          LD    D,H
9CB7 7D        0810          LD    A,L
9CB8 2D        0820          DEC   L
9CB9 B7        0830          OR    A
9CBA 200A      0840          JR    NZ,OK1
9CBC 7C        0850          LD    A,H
9CBD 25        0860          DEC   H
9CBE E607      0870          AND   7
9CC0 2004      0880          JR    NZ,OK1
9CC2 7C        0890          LD    A,H
9CC3 C608      0900          ADD   A,8
9CC5 67        0910          LD    H,A
9CC6 1A        0920 OK1      LD    A,(DE)
9CC7 CB3E      0930          SRL   (HL)
9CC9 3804      0940          JR    C,SETBIT
```

```
9CCB CB9F      0950         RES  3,A
9CCD 1802      0960         JR   BITOK
9CCF CBDF      0970 SETBIT  SET  3,A
9CD1 CB5E      0980 BITOK   BIT  3,(HL)
9CD3 2802      0990         JR   Z,BITSET
9CD5 CBFF      1000         SET  7,A
9CD7 12        1010 BITSET  LD   (DE),A
9CD8 10DB      1020         DJNZ BYTE1
9CDA CB9E      1030         RES  3,(HL)
9CDC E1        1040         POP  HL
9CDD 7C        1050         LD   A,H
9CDE C608      1060         ADD  A,8
9CE0 67        1070         LD   H,A
9CE1 C1        1080         POP  BC
9CE2 0D        1090         DEC  C
9CE3 20CC      1100         JR   NZ,LINE1
9CE5 7C        1110         LD   A,H
9CE6 D640      1120         SUB  64
9CE8 67        1130         LD   H,A
9CE9 7D        1140         LD   A,L
9CEA C650      1150         ADD  A,80
9CEC 6F        1160         LD   L,A
9CED 300A      1170         JR   NC,DONE1
9CEF 24        1180         INC  H
9CF0 7C        1190         LD   A,H
9CF1 E607      1200         AND  7
9CF3 2004      1210         JR   NZ,DONE1
```

```
9CF5 7C      1220        LD   A,H
9CF6 D608    1230        SUB  8
9CF8 67      1240        LD   H,A
9CF9 D1      1250 DONE1  POP  DE
9CFA 15      1260        DEC  D
9CFB 20B1    1270        JR   NZ,ROW1
9CFD C9      1280        RET
9CFE 05      1290 MODE0  DEC  B
9CFF D5      1300 ROW0   PUSH DE
9D00 0E08    1310        LD   C,8
9D02 C5      1320 LINE0  PUSH BC
9D03 E5      1330        PUSH HL
9D04 C5      1340 BYTE0  PUSH BC
9D05 5D      1350        LD   E,L
9D06 54      1360        LD   D,H
9D07 7D      1370        LD   A,L
9D08 2D      1380        DEC  L
9D09 B7      1390        OR   A
9D0A 200A    1400        JR   NZ,OK0
9D0C 7C      1410        LD   A,H
9D0D 25      1420        DEC  H
9D0E E607    1430        AND  7
9D10 2004    1440        JR   NZ,OK0
9D12 7C      1450        LD   A,H
9D13 C608    1460        ADD  A,8
9D15 67      1470        LD   H,A
```

```
9D16 7E        1480 OK0     LD   A,(HL)
9D17 17        1490         RLA
9D18 4F        1500         LD   C,A
9D19 1A        1510         LD   A,(DE)
9D1A 1F        1520         RRA
9D1B 1F        1530         RRA
9D1C 0604      1540         LD   B,4
9D1E 1F        1550 NXTROT  RRA
9D1F CB39      1560         SRL  C
9D21 CB39      1570         SRL  C
9D23 1F        1580         RRA
9D24 10F8      1590         DJNZ NXTROT
9D26 12        1600         LD   (DE),A
9D27 C1        1610         POP  BC
9D28 10DA      1620         DJNZ BYTE0
9D2A 7E        1630         LD   A,(HL)
9D2B 1F        1640         RRA
9D2C 0604      1650         LD   B,4
9D2E 0F        1660 LSTBYT  RRCA
9D2F CB3F      1670         SRL  A
9D31 10FB      1680         DJNZ LSTBYT
9D33 77        1690         LD   (HL),A
9D34 E1        1700         POP  HL
9D35 7C        1710         LD   A,H
9D36 C608      1720         ADD  A,8
9D38 67        1730         LD   H,A
9D39 C1        1740         POP  BC
```

```
9D3A 0D          1750          DEC   C
9D3B 20C5        1760          JR    NZ,LINE0
9D3D 7C          1770          LD    A,H
9D3E D640        1780          SUB   64
9D40 67          1790          LD    H,A
9D41 7D          1800          LD    A,L
9D42 C650        1810          ADD   A,80
9D44 6F          1820          LD    L,A
9D45 300A        1830          JR    NC,DONE0
9D47 24          1840          INC   H
9D48 7C          1850          LD    A,H
9D49 E607        1860          AND   7
9D4B 2004        1870          JR    NZ,DONE0
9D4D 7C          1880          LD    A,H
9D4E D608        1890          SUB   8
9D50 67          1900          LD    H,A
9D51 D1          1910 DONE0    POP   DE
9D52 15          1920          DEC   D
9D53 20AA        1930          JR    NZ,ROW0
9D55 C9          1940          RET
                 1950          END
```

Eleven - RSX Chords

The Amstrad is equipped with a marvellous feature, which can be used to create new commands which are then accepted as BASIC keywords. This feature is known as RSX and this routine uses this facility to create 24 new commands. Each of these commands plays a particular chord through the Amstrad's sound channels.

A chord, as I'm sure you know, is a combination of several notes in harmony, which are played simultaneously. Typing CALL *40000* will give you access to 24 of the more popular chords listed below (note that there are no spaces between parts of the new commands and that '#' equals sharp and b. equals flat):

New Command	Chord
|CMAJOR	C Major
|CMINOR	C Minor
|CSMAJOR	C# Major or Db. Major
|CSMINOR	C# Minor or Db. Minor
|DMAJOR	D Major
|DMINOR	D Minor
|DSMAJOR	D# Major or Eb. Major
|DSMINOR	D# Minor or Eb. Minor

```
|EMAJOR                    E Major
|EMINOR                    E Minor
|FMAJOR                    F Major
|FMINOR                    F Minor
|FSMAJOR                   F# Major or Gb. Majo
|FSMINOR                   F# Minor or Gb. Mino
|GMAJOR                    G Major
|GMINOR                    G Minor
|GSMAJOR                   G# Major or Ab. Majo
|GSMINOR                   G# Minor or Ab. Mino
|AMAJOR                    A Major
|AMINOR                    A Minor
|ASMAJOR                   A# Major or Bb. Majo
|ASMINOR                   A# Minor or Bb. Mino
|BMAJOR                    B Major
|BMINOR                    B Minor
```

All of the chords are made up of three notes, and use the three Amstrad sound channels to produce the chord.

To get a chord, you simply type in one of the commands. It is a good idea to put the volume control at around three quarters of maximum as with all multi-channel sound on the Amstrad, the sound output is a bit too powerful for the built-in speaker to handle without creating a fair amount of distortion.

```
1 '                    CHORDS
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 240
20 FOR n=40000 TO 40617
30 READ a$:POKE n,VAL("&"+a$)
40 NEXT
50 DATA 01,4A,9C,21,94,9C,CD,D1,BC,C9
60 DATA 98,9C,C3,33,9D,C3,37,9D,C3,3B
70 DATA 9D,C3,3F,9D,C3,43,9D,C3,47,9D
80 DATA C3,4B,9D,C3,4F,9D,C3,53,9D,C3
90 DATA 57,9D,C3,5B,9D,C3,5F,9D,C3,63
100 DATA 9D,C3,67,9D,C3,6B,9D,C3,6F,9D
110 DATA C3,73,9D,C3,77,9D,C3,7B,9D,C3
120 DATA 7F,9D,C3,83,9D,C3,87,9D,C3,8B
130 DATA 9D,C3,8F,9D,00,00,00,00,43,4D
140 DATA 41,4A,4F,D2,43,4D,49,4E,4F,D2
150 DATA 43,53,4D,41,4A,4F,D2,43,53,4D
160 DATA 49,4E,4F,D2,44,4D,41,4A,4F,D2
170 DATA 44,4D,49,4E,4F,D2,44,53,4D,41
180 DATA 4A,4F,D2,44,53,4D,49,4E,4F,D2
190 DATA 45,4D,41,4A,4F,D2,45,4D,49,4E
200 DATA 4F,D2,46,4D,41,4A,4F,D2,46,4D
```

```
210 DATA 49,4E,4F,D2,46,53,4D,41,4A,4F
220 DATA D2,46,53,4D,49,4E,4F,D2,47,4D
230 DATA 41,4A,4F,D2,47,4D,49,4E,4F,D2
240 DATA 47,53,4D,41,4A,4F,D2,47,53,4D
250 DATA 49,4E,4F,D2,41,4D,41,4A,4F,D2
260 DATA 41,4D,49,4E,4F,D2,41,53,4D,41
270 DATA 4A,4F,D2,41,53,4D,49,4E,4F,D2
280 DATA 42,4D,41,4A,4F,D2,42,4D,49,4E
290 DATA 4F,D2,00,1E,00,18,5C,1E,06,18
300 DATA 58,1E,0C,18,54,1E,12,18,50,1E
310 DATA 18,18,4C,1E,1E,18,48,1E,24,18
320 DATA 44,1E,2A,18,40,1E,30,18,3C,1E
330 DATA 36,18,38,1E,3C,18,34,1E,42,18
340 DATA 30,1E,48,18,2C,1E,4E,18,28,1E
350 DATA 54,18,24,1E,5A,18,20,1E,60,18
360 DATA 1C,1E,66,18,18,1E,6C,18,14,1E
370 DATA 72,18,10,1E,78,18,0C,1E,7E,18
380 DATA 08,1E,84,18,04,1E,8A,18,00,21
390 DATA 1A,9E,16,00,19,5E,23,56,23,ED
400 DATA 53,02,9E,5E,23,56,23,ED,53,0B
410 DATA 9E,5E,23,56,ED,53,14,9E,F5,B7
420 DATA 28,03,DD,7E,00,32,00,9E,32,09
430 DATA 9E,32,12,9E,B7,28,12,CD,C2,BC
440 DATA D0,23,23,7E,B7,28,08,CB,7F,20
450 DATA 04,3E,00,18,02,3E,0F,32,05,9E
460 DATA 32,0E,9E,32,17,9E,F1,FE,02,CC
```

```
470 DATA A7,BC,21,FF,9D,CD,AA,BC,30,FB
480 DATA 21,08,9E,CD,AA,BC,30,FB,21,11
490 DATA 9E,CD,AA,BC,30,FB,C9,31,00,00
500 DATA 00,00,00,00,00,00,2A,00,00,00
510 DATA 00,00,00,00,00,1C,00,00,00,00
520 DATA 00,00,00,00,7E,02,DE,01,7B,01
530 DATA 7E,02,DE,01,92,01,5A,02,C3,01
540 DATA 66,01,5A,02,C3,01,7B,01,A4,02
550 DATA 38,02,AA,01,38,02,AA,01,66,01
560 DATA 7E,02,18,02,92,01,A4,02,18,02
570 DATA 92,01,5A,02,FA,01,7B,01,7E,02
580 DATA FA,01,7B,01,38,02,DE,01,66,01
590 DATA 5A,02,DE,01,66,01,A4,02,18,02
600 DATA C3,01,A4,02,38,02,C3,01,7E,02
610 DATA FA,01,AA,01,7E,02,18,02,AA,01
620 DATA 5A,02,DE,01,92,01,5A,02,FA,01
630 DATA 92,01,38,02,C3,01,7B,01,38,02
640 DATA DE,01,7B,01,18,02,AA,01,66,01
650 DATA 18,02,C3,01,66,01,A4,02,FA,01
660 DATA 92,01,A4,02,FA,01,AA,01
```

Envelopes

Let's move onto the more complex part of using these chords, using them with sound envelopes. If you 'play' the basic chord without additional parameters, you get a basic "organ-like" chord. The routine automatically sets the command to envelope Ø which makes each chord last for about two seconds.

It is possible to create your own defined envelopes which can be incorporated within the chord sound. To do this, set up a volume envelope as normal (if you are not familiar with the Amstrad's sound features then check the user manual). Next, assign the envelope number to the end of the chord command. For example, if your envelope was ENV 1,3,5,1,15,-1,1Ø with the first number corresponding to the envelope's number then you would add '1' to the end of the chord command, i.e. |CMAJOR, 1.

The initial volume for the chord is dependent upon the envelope being used. It is assessed by the program in the following way. If the step size in the first section of the envelope being used is positive (excluding Ø) then the initial volume is set to Ø. If it is negative then the initial volume is set to its maximum value, 15. This gives the envelope the greatest volume range to work with.

Using an envelope step size of Ø can make the chord play for as long as you wish. For example:

```
ENV 2,1Ø,Ø,1ØØ
|CMAJOR,2
```

This plays a C Major chord for ten seconds while ENV 2,n,Ø,1ØØ would play the same chord for n seconds.

When the commands are executed the routine will wait until all three of the sound buffers are available before adding the notes that make up the chord to the sound queues. If you want the sound queues to be cleared and the chord to sound immediately when the command is executed then add an extra parameter **before** the envelope number, e.g.

```
ENV 4,15,-1,2Ø
|CMAJOR,Ø,4
|DMAJOR,Ø,4
```

In this example, only the D Major chord would sound as the first chord would be cut off as soon as it began. **Any** value can be used for this extra first parameter. The routine only scans for the presence of an extra parameter, and not for the value of that parameter.

Obviously, you may not be happy at the thought of losing your first chord immediately. You can get around this simply by setting up a delay loop between each chord. An example is given below:

```
ENV 5,8,Ø,1ØØ
|EMAJOR,Ø,5
FOR T=1 TO 3ØØ:NEXT
|CMAJOR Ø,5
FOR T=1 TO 25Ø:NEXT
|DMAJOR,Ø,5
```

```
                    0001 ;              CHORDS
                    0002 ;
9C40                0010          ORG   40000
BCA7                0020 SNDRES   DEFL  0BCA7H
BCAA                0030 ADDSND   DEFL  0BCAAH
BCC2                0040 AMPADR   DEFL  0BCC2H
BCD1                0050 LOGEXT   DEFL  0BCD1H
9C40 014A9C         0060          LD    BC,TABLE
9C43 21949C         0070          LD    HL,SPACE
9C46 CDD1BC         0080          CALL  LOGEXT
9C49 C9             0090          RET
9C4A 989C           0100 TABLE    DEFW  NAMETB
9C4C C3339D         0110          JP    CMAJ
9C4F C3379D         0120          JP    CMIN
9C52 C33B9D         0130          JP    CSMAJ
```

```
9C55 C33F9D    0140          JP    CSMIN
9C58 C3439D    0150          JP    DMAJ
9C5B C3479D    0160          JP    DMIN
9C5E C34B9D    0170          JP    DSMAJ
9C61 C34F9D    0180          JP    DSMIN
9C64 C3539D    0190          JP    EMAJ
9C67 C3579D    0200          JP    EMIN
9C6A C35B9D    0210          JP    FMAJ
9C6D C35F9D    0220          JP    FMIN
9C70 C3639D    0230          JP    FSMAJ
9C73 C3679D    0240          JP    FSMIN
9C76 C36B9D    0250          JP    GMAJ
9C79 C36F9D    0260          JP    GMIN
9C7C C3739D    0270          JP    GSMAJ
9C7F C3779D    0280          JP    GSMIN
9C82 C37B9D    0290          JP    AMAJ
9C85 C37F9D    0300          JP    AMIN
9C88 C3839D    0310          JP    ASMAJ
9C8B C3879D    0320          JP    ASMIN
9C8E C38B9D    0330          JP    BMAJ
9C91 C38F9D    0340          JP    BMIN
9C94 00        0350 SPACE    DEFB  0,0,0,0
     00 00 00
9C98           0360 NAMETB   DEFM  "CMAJO"
9C9D D2        0370          DEFB  "R"+80H
9C9E           0380          DEFM  "CMINO"
```

```
9CA3 D2          0390          DEFB "R"+80H
9CA4             0400          DEFM "CSMAJO"
9CAA D2          0410          DEFB "R"+80H
9CAB             0420          DEFM "CSMINO"
9CB1 D2          0430          DEFB "R"+80H
9CB2             0440          DEFM "DMAJO"
9CB7 D2          0450          DEFB "R"+80H
9CB8             0460          DEFM "DMINO"
9CBD D2          0470          DEFB "R"+80H
9CBE             0480          DEFM "DSMAJO"
9CC4 D2          0490          DEFB "R"+80H
9CC5             0500          DEFM "DSMINO"
9CCB D2          0510          DEFB "R"+80H
9CCC             0520          DEFM "EMAJO"
9CD1 D2          0530          DEFB "R"+80H
9CD2             0540          DEFM "EMINO"
9CD7 D2          0550          DEFB "R"+80H
9CD8             0560          DEFM "FMAJO"
9CDD D2          0570          DEFB "R"+80H
9CDE             0580          DEFM "FMINO"
9CE3 D2          0590          DEFB "R"+80H
9CE4             0600          DEFM "FSMAJO"
9CEA D2          0610          DEFB "R"+80H
9CEB             0620          DEFM "FSMINO"
9CF1 D2          0630          DEFB "R"+80H
9CF2             0640          DEFM "GMAJO"
9CF7 D2          0650          DEFB "R"+80H
```

```
9CF8            0660          DEFM "GMINO"
9CFD D2         0670          DEFB "R"+80H
9CFE            0680          DEFM "GSMAJO"
9D04 D2         0690          DEFB "R"+80H
9D05            0700          DEFM "GSMINO"
9D0B D2         0710          DEFB "R"+80H
9D0C            0720          DEFM "AMAJO"
9D11 D2         0730          DEFB "R"+80H
9D12            0740          DEFM "AMINO"
9D17 D2         0750          DEFB "R"+80H
9D18            0760          DEFM "ASMAJO"
9D1E D2         0770          DEFB "R"+80H
9D1F            0780          DEFM "ASMINO"
9D25 D2         0790          DEFB "R"+80H
9D26            0800          DEFM "BMAJO"
9D2B D2         0810          DEFB "R"+80H
9D2C            0820          DEFM "BMINO"
9D31 D2         0830          DEFB "R"+80H
9D32 00         0840          DEFB 0
9D33 1E00       0850 CMAJ     LD   E,0
9D35 185C       0860          JR   CHORD
9D37 1E06       0870 CMIN     LD   E,6
9D39 1858       0880          JR   CHORD
9D3B 1E0C       0890 CSMAJ    LD   E,12
9D3D 1854       0900          JR   CHORD
9D3F 1E12       0910 CSMIN    LD   E,18
9D41 1850       0920          JR   CHORD
```

```
9D43 1E18      0930 DMAJ    LD    E,24
9D45 184C      0940         JR    CHORD
9D47 1E1E      0950 DMIN    LD    E,30
9D49 1848      0960         JR    CHORD
9D4B 1E24      0970 DSMAJ   LD    E,36
9D4D 1844      0980         JR    CHORD
9D4F 1E2A      0990 DSMIN   LD    E,42
9D51 1840      1000         JR    CHORD
9D53 1E30      1010 EMAJ    LD    E,48
9D55 183C      1020         JR    CHORD
9D57 1E36      1030 EMIN    LD    E,54
9D59 1838      1040         JR    CHORD
9D5B 1E3C      1050 FMAJ    LD    E,60
9D5D 1834      1060         JR    CHORD
9D5F 1E42      1070 FMIN    LD    E,66
9D61 1830      1080         JR    CHORD
9D63 1E48      1090 FSMAJ   LD    E,72
9D65 182C      1100         JR    CHORD
9D67 1E4E      1110 FSMIN   LD    E,78
9D69 1828      1120         JR    CHORD
9D6B 1E54      1130 GMAJ    LD    E,84
9D6D 1824      1140         JR    CHORD
9D6F 1E5A      1150 GMIN    LD    E,90
9D71 1820      1160         JR    CHORD
9D73 1E60      1170 GSMAJ   LD    E,96
9D75 181C      1180         JR    CHORD
```

```
9D77 1E66      1190 GSMIN  LD   E,102
9D79 1818      1200        JR   CHORD
9D7B 1E6C      1210 AMAJ   LD   E,108
9D7D 1814      1220        JR   CHORD
9D7F 1E72      1230 AMIN   LD   E,114
9D81 1810      1240        JR   CHORD
9D83 1E78      1250 ASMAJ  LD   E,120
9D85 180C      1260        JR   CHORD
9D87 1E7E      1270 ASMIN  LD   E,126
9D89 1808      1280        JR   CHORD
9D8B 1E84      1290 BMAJ   LD   E,132
9D8D 1804      1300        JR   CHORD
9D8F 1E8A      1310 BMIN   LD   E,138
9D91 1800      1320        JR   CHORD
9D93 211A9E    1330 CHORD  LD   HL,DATA
9D96 1600      1340        LD   D,0
9D98 19        1350        ADD  HL,DE
9D99 5E        1360        LD   E,(HL)
9D9A 23        1370        INC  HL
9D9B 56        1380        LD   D,(HL)
9D9C 23        1390        INC  HL
9D9D ED53029E  1400        LD   (TONE1),DE
9DA1 5E        1410        LD   E,(HL)
9DA2 23        1420        INC  HL
9DA3 56        1430        LD   D,(HL)
9DA4 23        1440        INC  HL
```

```
9DA5 ED530B9E 1450         LD   (TONE2),DE
9DA9 5E       1460         LD   E,(HL)
9DAA 23       1470         INC  HL
9DAB 56       1480         LD   D,(HL)
9DAC ED53149E 1490         LD   (TONE3),DE
9DB0 F5       1500         PUSH AF
9DB1 B7       1510         OR   A
9DB2 2803     1520         JR   Z,SETENV
9DB4 DD7E00   1530         LD   A,(IX+0)
9DB7 32009E   1540 SETENV  LD   (ENV1),A
9DBA 32099E   1550         LD   (ENV2),A
9DBD 32129E   1560         LD   (ENV3),A
9DC0 B7       1570         OR   A
9DC1 2812     1580         JR   Z,NEGTVE
9DC3 CDC2BC   1590 FNDENV  CALL AMPADR
9DC6 D0       1600         RET  NC
9DC7 23       1610         INC  HL
9DC8 23       1620         INC  HL
9DC9 7E       1630         LD   A,(HL)
9DCA B7       1640         OR   A
9DCB 2808     1650         JR   Z,NEGTVE
9DCD CB7F     1660         BIT  7,A
9DCF 2004     1670         JR   NZ,NEGTVE
9DD1 3E00     1680         LD   A,0
9DD3 1802     1690         JR   SETAMP
9DD5 3E0F     1700 NEGTVE  LD   A,15
9DD7 32059E   1710 SETAMP  LD   (AMPL1),A
```

```
9DDA 320E9E    1720          LD    (AMPL2),A
9DDD 32179E    1730          LD    (AMPL3),A
9DE0 F1        1740          POP   AF
9DE1 FE02      1750          CP    2
9DE3 CCA7BC    1760          CALL  Z,SNDRES
9DE6 21FF9D    1770          LD    HL,SND1
9DE9 CDAABC    1780 AFULL    CALL  ADDSND
9DEC 30FB      1790          JR    NC,AFULL
9DEE 21089E    1800          LD    HL,SND2
9DF1 CDAABC    1810 BFULL    CALL  ADDSND
9DF4 30FB      1820          JR    NC,BFULL
9DF6 21119E    1830          LD    HL,SND3
9DF9 CDAABC    1840 CFULL    CALL  ADDSND
9DFC 30FB      1850          JR    NC,CFULL
9DFE C9        1860          RET
9DFF 31        1870 SND1     DEFB  49
9E00 00        1880 ENV1     DEFB  0,0
     00
9E02 00        1890 TONE1    DEFB  0,0,0
     00 00
9E05 00        1900 AMPL1    DEFB  0,0,0
     00 00
9E08 2A        1910 SND2     DEFB  42
9E09 00        1920 ENV2     DEFB  0,0
     00
9E0B 00        1930 TONE2    DEFB  0,0,0
     00 00
```

```
9E0E 00          1940 AMPL2   DEFB 0,0,0
     00 00
9E11 1C          1950 SND3    DEFB 28
9E12 00          1960 ENV3    DEFB 0,0
     00
9E14 00          1970 TONE3   DEFB 0,0,0
     00 00
9E17 00          1980 AMPL3   DEFB 0,0,0
     00 00
9E1A 7E02        1990 DATA    DEFW 638,478,379
     DE01 7B01
9E20 7E02        2000         DEFW 638,478,402
     DE01 9201
9E26 5A02        2010         DEFW 602,451,358
     C301 6601
9E2C 5A02        2020         DEFW 602,451,379
     C301 7B01
9E32 A402        2030         DEFW 676,568,426
     3802 AA01
9E38 3802        2040         DEFW 568,426,358
     AA01 6601
9E3E 7E02        2050         DEFW 638,536,402
     1802 9201
9E44 A402        2060         DEFW 676,536,402
     1802 9201
9E4A 5A02        2070         DEFW 602,506,379
     FA01 7B01
```

```
9E50 7E02      2080          DEFW 638,506,379
     FA01 7B01
9E56 3802      2090          DEFW 568,478,358
     DE01 6601
9E5C 5A02      2100          DEFW 602,478,358
     DE01 6601
9E62 A402      2110          DEFW 676,536,451
     1802 C301
9E68 A402      2120          DEFW 676,568,451
     3802 C301
9E6E 7E02      2130          DEFW 638,506,426
     FA01 AA01
9E74 7E02      2140          DEFW 638,536,426
     1802 AA01
9E7A 5A02      2150          DEFW 602,478,402
     DE01 9201
9E80 5A02      2160          DEFW 602,506,402
     FA01 9201
9E86 3802      2170          DEFW 568,451,379
     C301 7B01
9E8C 3802      2180          DEFW 568,478,379
     DE01 7B01
9E92 1802      2190          DEFW 536,426,358
     AA01 6601
9E98 1802      2200          DEFW 536,451,358
     C301 6601
```

```
9E9E A402      2210          DEFW 676,506,402
     FA01 9201
9EA4 A402      2220          DEFW 676,506,426
     FA01 AA01
               2230          END
```

Chord Organ

We couldn't leave this routine without providing a chord organ program. It is not, however, just a matter of using lines like IF INKEY$="A" THEN |AMAJOR, as there are other matters such as interrupts and envelopes which have to be controlled. Our program plays all the major chords, natural notes, sharps and flats.

Here is the 'keyboard':

	2	3		5	6	7	
Q	W	E	R	T	Y	U	I

And these are the notes they represent:

	C#	D#		F#	G#	A#	
C	D	E	F	G	A	B	C

```
10 ENV 1,10,0,100
20 DIM k(12):m=-1
30 FOR n=0 TO 12:READ k(n):NEXT
40 n=0
```

```
50 a=INKEY(k(n))
60 IF a=0 AND n=m GOTO 40
70 IF a=0 GOTO 120
80 n=n+1
90 IF n<13 GOTO 50
100 SOUND 135,0,0,0:m=-1
110 GOTO 40
120 ON n+1 GOSUB 140,150,160,170,180,190
,200,210,220,230,240,250,260
130 m=n:GOTO 40
140 |CMAJOR,0,1:RETURN
150 |DMAJOR,0,1:RETURN
160 |EMAJOR,0,1:RETURN
170 |FMAJOR,0,1:RETURN
180 |GMAJOR,0,1:RETURN
190 |AMAJOR,0,1:RETURN
200 |BMAJOR,0,1:RETURN
210 |CMAJOR,0,1:RETURN
220 |CSMAJOR,0,1:RETURN
230 |DSMAJOR,0,1:RETURN
240 |FSMAJOR,0,1:RETURN
250 |GSMAJOR,0,1:RETURN
260 |ASMAJOR,0,1:RETURN
270 DATA 67,59,58,50,51,43,42,35,65,57,4
9,48,41
```

Twelve – Screen Compactors

Compactor.1

The Amstrad's screen is stored in 16K of memory. This can be a lot of RAM to store what is often a simple screen display. Much of a screen is usually blank and has a value of Ø. A blank section of the screen which is 2Ø pixels is represented in the screen RAM by twenty zeros. This is a lot of memory wasted. Our compactor routine makes much better use of the screen memory.

This program works by first looking at a section of screen RAM. If it has a value other than Ø, then it stores that value as it is without alteration. If the value is Ø, it counts how many zeros follow and then stores one zero and a second value equalling the number of zeros.

```
1 '              COMPACT
10 SYMBOL AFTER 256:MEMORY 20000:SYMBOL
AFTER 240
20 FOR n=43800 TO 43888
30 READ a$:POKE n,VAL("&"+a$)
40 NEXT
50 DATA 11,00,C0,21,17,AB,7E,12,13,2B
60 DATA CB,7A,C8,B7,20,F6,46,2B,05,28
70 DATA F1,12,13,10,FC,CB,7A,20,E9,C9
80 DATA FE,01,C0,11,00,C0,21,17,AB,1A
90 DATA 77,13,2B,CB,7A,28,18,B7,20,F5
```

```
100 DATA 06,01,1A,B7,20,08,13,CB,7A,28
110 DATA 07,04,20,F4,70,2B,18,E3,04,70
120 DATA 2B,EB,21,17,AB,B7,ED,52,EB,DD
130 DATA 6E,00,DD,66,01,73,23,72,C9
```

To compact a screen, use the following:

A%=Ø: CALL 4383Ø, @A%

The number of bytes taken up by the screen is returned to the A% variable, so that you can check on the memory used and thus calculate the approximate memory saving. (This also applies to the second compactor routine following this one.)

To de-compact and display a previously compacted screen, type:

CALL 438ØØ

To save the screen, firstly compact it in the normal way and then type:

SAVE "NAME",B,438ØØ-A%, A%+3Ø

To load and display, simply LOAD "" and then CALL 438ØØ.

When saving the screen, you have to save the de-compactor routine as well.

```
                0001 ;              COMPACT1
                0002 ;
AB18            0010        ORG     43800
                0020 ;              ROUTINE1
```

```
AB18 1100C0    0030          LD    DE,0C000H
AB1B 2117AB    0040          LD    HL,43799
AB1E 7E        0050 UNCMPT   LD    A,(HL)
AB1F 12        0060          LD    (DE),A
AB20 13        0070          INC   DE
AB21 2B        0080          DEC   HL
AB22 CB7A      0090          BIT   7,D
AB24 C8        0100          RET   Z
AB25 B7        0110          OR    A
AB26 20F6      0120          JR    NZ,UNCMPT
AB28 46        0130          LD    B,(HL)
AB29 2B        0140          DEC   HL
AB2A 05        0150          DEC   B
AB2B 28F1      0160          JR    Z,UNCMPT
AB2D 12        0170 REPEAT   LD    (DE),A
AB2E 13        0180          INC   DE
AB2F 10FC      0190          DJNZ  REPEAT
AB31 CB7A      0200          BIT   7,D
AB33 20E9      0210          JR    NZ,UNCMPT
AB35 C9        0220          RET
               0230 ;              ROUTINE2
AB36 FE01      0240          CP    1
AB38 C0        0250          RET   NZ
AB39 1100C0    0260          LD    DE,0C000H
AB3C 2117AB    0270          LD    HL,43799
AB3F 1A        0280 COMPCT   LD    A,(DE)
AB40 77        0290          LD    (HL),A
```

```
AB41 13       0300        INC  DE
AB42 2B       0310        DEC  HL
AB43 CB7A     0320        BIT  7,D
AB45 2818     0330        JR   Z,TOTAL
AB47 B7       0340        OR   A
AB48 20F5     0350        JR   NZ,COMPCT
AB4A 0601     0360        LD   B,1
AB4C 1A       0370 COUNT  LD   A,(DE)
AB4D B7       0380        OR   A
AB4E 2008     0390        JR   NZ,GOTLEN
AB50 13       0400        INC  DE
AB51 CB7A     0410        BIT  7,D
AB53 2807     0420        JR   Z,LAST
AB55 04       0430        INC  B
AB56 20F4     0440        JR   NZ,COUNT
AB58 70       0450 GOTLEN LD   (HL),B
AB59 2B       0460        DEC  HL
AB5A 18E3     0470        JR   COMPCT
AB5C 04       0480 LAST   INC  B
AB5D 70       0490        LD   (HL),B
AB5E 2B       0500        DEC  HL
AB5F EB       0510 TOTAL  EX   DE,HL
AB60 2117AB   0520        LD   HL,43799
AB63 B7       0530        OR   A
AB64 ED52     0540        SBC  HL,DE
AB66 EB       0550        EX   DE,HL
```

```
AB67 DD6E00     0560        LD    L,(IX+0)
AB6A DD6601     0570        LD    H,(IX+1)
AB6D 73         0580        LD    (HL),E
AB6E 23         0590        INC   HL
AB6F 72         0600        LD    (HL),D
AB70 C9         0610        RET
                0620        END
```

Compactor.2

Unlike the previous routine, this one compacts **all** multiples of the same value and not just zeros. If, for example, a portion of the screen read as:

3, 16Ø, 2, 2, 2, 2, 2, 2, 8, 8, 8, 8

This routine stores it as:

3, 1, 16Ø, 1, 2, 6, 8, 4

The major drawback with this routine is that if a screen is too complicated then it doesn't compact at all and may take even more than the 16K used for storing the screen in normal circumstances. However, as it will work with most screens, you should always give it a try first.

```
1 '                COMPACT
10 SYMBOL AFTER 256:MEMORY 20000:SYMBOL
AFTER 240
20 FOR n=43800 TO 43875
30 READ a$:POKE n,VAL("&"+a$)
```

```
40 NEXT
50 DATA 11,00,C0,21,17,AB,7E,2B,46,2B
60 DATA 12,13,10,FC,CB,7A,20,F4,C9,FE
70 DATA 01,C0,11,00,C0,21,17,AB,1A,77
80 DATA 13,2B,06,01,CB,7A,28,12,4F,1A
90 DATA B9,20,08,13,CB,7A,28,07,04,20
100 DATA F4,70,2B,18,E5,04,70,2B,EB,21
110 DATA 17,AB,B7,ED,52,EB,DD,6E,00,DD
120 DATA 66,01,73,23,72,C9
```

To compact a screen:

A%=Ø:CALL 43819, @A%

To de-compact a screen:

CALL 438ØØ

To save the screen and the uncompactor (which is needed for displaying the screen in its normal way) type:

SAVE "NAME",B,438ØØ-A%,A%+19

B is not a variable in this case of course, but is the sign telling the computer that you are saving a binary file.

```
                0001 ;              COMPACT2
                0002 ;
AB18            0010          ORG   43800
                0020 ;              ROUTINE1
```

```
AB18 1100C0    0030          LD    DE,0C000H
AB1B 2117AB    0040          LD    HL,43799
AB1E 7E        0050 UNCMPT   LD    A,(HL)
AB1F 2B        0060          DEC   HL
AB20 46        0070          LD    B,(HL)
AB21 2B        0080          DEC   HL
AB22 12        0090 REPEAT   LD    (DE),A
AB23 13        0100          INC   DE
AB24 10FC      0110          DJNZ  REPEAT
AB26 CB7A      0120          BIT   7,D
AB28 20F4      0130          JR    NZ,UNCMPT
AB2A C9        0140          RET
               0150 ;              ROUTINE2
AB2B FE01      0160          CP    1
AB2D C0        0170          RET   NZ
AB2E 1100C0    0180          LD    DE,0C000H
AB31 2117AB    0190          LD    HL,43799
AB34 1A        0200 COMPCT   LD    A,(DE)
AB35 77        0210          LD    (HL),A
AB36 13        0220          INC   DE
AB37 2B        0230          DEC   HL
AB38 0601      0240          LD    B,1
AB3A CB7A      0250          BIT   7,D
AB3C 2812      0260          JR    Z,TOTAL
AB3E 4F        0270          LD    C,A
AB3F 1A        0280 COUNT    LD    A,(DE)
AB40 B9        0290          CP    C
```

```
AB41 2008      0300         JR    NZ,GOTLEN
AB43 13        0310         INC   DE
AB44 CB7A      0320         BIT   7,D
AB46 2807      0330         JR    Z,LAST
AB48 04        0340         INC   B
AB49 20F4      0350         JR    NZ,COUNT
AB4B 70        0360 GOTLEN  LD    (HL),B
AB4C 2B        0370         DEC   HL
AB4D 18E5      0380         JR    COMPCT
AB4F 04        0390 LAST    INC   B
AB50 70        0400 TOTAL   LD    (HL),B
AB51 2B        0410         DEC   HL
AB52 EB        0420         EX    DE,HL
AB53 2117AB    0430         LD    HL,43799
AB56 B7        0440         OR    A
AB57 ED52      0450         SBC   HL,DE
AB59 EB        0460         EX    DE,HL
AB5A DD6E00    0470         LD    L,(IX+0)
AB5D DD6601    0480         LD    H,(IX+1)
AB60 73        0490         LD    (HL),E
AB61 23        0500         INC   HL
AB62 72        0510         LD    (HL),D
AB63 C9        0520         RET
               0530         END
```

Thirteen – DOKE and DEEK

In machine code programming, you often have to POKE and PEEK 16-bit numbers. Whereas 8-bit numbers have a range of 256, 16-bit numbers have a range of 65536. 16-bit numbers use two bytes and to POKE a two byte number, you have to use a little formula:

POKE byte 1 + 256*byte 2

This also applies to PEEKing two bytes of a memory location. Some computers are fortunate enough to have Double-PEEK and Double-POKE commands, known as DEEK and DOKE. This routine provides you with two new BASIC commands, |DOKE and |DEEK.

```
1 '                DOKE/DEEK
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 240
20 FOR n=40000 TO 40071
30 READ a$:POKE n,VAL("&"+a$)
40 NEXT
50 DATA 01,4A,9C,21,52,9C,CD,D1,BC,C9
60 DATA 56,9C,C3,5F,9C,C3,72,9C,00,00
70 DATA 00,00,44,4F,4B,C5,44,45,45,CB
80 DATA 00,FE,02,C0,DD,6E,02,DD,66,03
90 DATA DD,7E,00,77,23,DD,7E,01,77,C9
```

```
100 DATA FE,02,C0,DD,6E,02,DD,66,03,DD
110 DATA 5E,00,DD,56,01,7E,23,12,13,7E
120 DATA 12,C9
```

You must first CALL 40000 to set up the new commands. To use |DOKE you need to follow it with a valid memory address and a 16-bit number. Both of these values should be in decimal. |DOKE, 30000,343 would split the value 343 up using the reverse of the formula above and place the two values into memory locations 30000 and 30001.

|DEEK is slightly more complicated. You use it in the format:

N%=0:|DEEK, A, @N%

A stands for the first address to be PEEKed (|DEEK will look at A and A+1) while N% contains the value returned by the |DEEK. N% must be defined before a |DEEK. As you can see from the percent sign, it must be an integer variable. Remember this if you want to print out the actual value of DEEK. If the value returned is negative, you need to print 65536+N% to get the true value.

```
                    0001 ;          DOKE DEEK
                    0002 ;
9C40                0010            ORG  40000
BCD1                0020 LOGEXT     DEFL 0BCD1H
9C40 014A9C         0030            LD   BC,TABLE
9C43 21529C         0040            LD   HL,SPACE
9C46 CDD1BC         0050            CALL LOGEXT
9C49 C9             0060            RET
9C4A 569C           0070 TABLE      DEFW NAMETB
```

```
9C4C C35F9C    0080          JP    DOKE
9C4F C3729C    0090          JP    DEEK
9C52 00        0100 SPACE    DEFB  0,0,0,0
     00 00 00
9C56           0110 NAMETB   DEFM  "DOK"
9C59 C5        0120          DEFB  "E"+80H
9C5A           0130          DEFM  "DEE"
9C5D CB        0140          DEFB  "K"+80H
9C5E 00        0150          DEFB  0
9C5F FE02      0160 DOKE     CP    2
9C61 C0        0170          RET   NZ
9C62 DD6E02    0180          LD    L,(IX+2)
9C65 DD6603    0190          LD    H,(IX+3)
9C68 DD7E00    0200          LD    A,(IX+0)
9C6B 77        0210          LD    (HL),A
9C6C 23        0220          INC   HL
9C6D DD7E01    0230          LD    A,(IX+1)
9C70 77        0240          LD    (HL),A
9C71 C9        0250          RET
9C72 FE02      0260 DEEK     CP    2
9C74 C0        0270          RET   NZ
9C75 DD6E02    0280          LD    L,(IX+2)
9C78 DD6603    0290          LD    H,(IX+3)
9C7B DD5E00    0300          LD    E,(IX+0)
9C7E DD5601    0310          LD    D,(IX+1)
9C81 7E        0320          LD    A,(HL)
```

```
9C82 23          0330          INC   HL
9C83 12          0340          LD    (DE),A
9C84 13          0350          INC   DE
9C85 7E          0360          LD    A,(HL)
9C86 12          0370          LD    (DE),A
9C87 C9          0380          RET
                 0390          END
```

Fourteen – The Games Writing Package

Finally in this book we have a program which gives you a selection of machine code routines all in one easily-entered package. The routines create new BASIC keywords, using RSX calls.

|EXPLODE, |READCHAR, |BIGPRINT, |USCROLL, |DSCROLL, |LSCROLL, |RSCROLL, |COMMANDS

Let's deal with each in turn. EXPLODE needn't be followed by any parameters. It simply creates an explosion sound of great use in games programs. READCHAR and BIGPRINT are two commands corresponding to the titles of routines elsewhere in this book. BIGPRINT creates double-size characters while READCHAR calculates what ASCII character is in a specified screen position.

USCROLL, DSCROLL, LSCROLL and RSCROLL are the four box scrolls featured in this routines section with the first letter standing for the direction of the scroll. COMMANDS simply gives a list of all the above available commands.

To understand how each feature works and the parameters which must follow the command word, you'll need to turn to the appropriate individual routine.

The features in this package offer most of the machine code routines needed to write a good "hybrid" game, partly in BASIC and partly in machine code.

```
1 '        GAMES WRITING PACKAGE
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 240
20 DIM x(12)
30 FOR c=0 TO 12:READ a$:x(c)=VAL("&"+a$
):NEXT
40 c=0:sum=0
50 FOR n=40000 TO 41207
60 READ a$:v=VAL("&"+a$)
70 sum=sum+v:POKE n,v
80 IF (n+1-40000) MOD 100<>0 GOTO 110
90 IF sum<>x(c) THEN PRINT"*ERROR IN DAT
A*";CHR$(7):PRINT"Check lines";210+100*c
;"to";300+100*c:END
100 sum=0:c=c+1
110 NEXT
120 IF sum<>x(c) THEN PRINT"*ERROR IN DA
TA*";CHR$(7):PRINT"Check line 1410"
130 '
140 '            CHECKSUMS
150 '
160 DATA 2B79,2F09,2A7A,27A3,263B,2434,2
C42
170 DATA 2782,2593,2A89,278C,29D3,04DD
180 '
190 '           MACHINE CODE
```

```
200 '
210 DATA 01,4A,9C,21,64,9C,CD,D1,BC,C9
220 DATA 68,9C,C3,A4,9C,C3,C3,9C,C3,D7
230 DATA 9C,C3,77,9D,C3,09,9E,C3,A1,9E
240 DATA C3,B1,9F,C3,C7,A0,00,00,00,00
250 DATA 45,58,50,4C,4F,44,C5,52,45,41
260 DATA 44,43,48,41,D2,42,49,47,50,52
270 DATA 49,4E,D4,55,53,43,52,4F,4C,CC
280 DATA 44,53,43,52,4F,4C,CC,4C,53,43
290 DATA 52,4F,4C,CC,52,53,43,52,4F,4C
300 DATA CC,43,4F,4D,4D,41,4E,44,D3,00
310 DATA CD,A7,BC,3E,01,21,B6,9C,CD,BC
320 DATA BC,21,BA,9C,CD,AA,BC,C9,01,0F
330 DATA FF,19,01,01,00,00,00,0F,0F,00
340 DATA 00,DD,6E,02,DD,66,04,CD,75,BB
350 DATA CD,60,BB,DD,6E,00,DD,66,01,77
360 DATA C9,CD,93,BB,F5,DD,6E,04,DD,66
370 DATA 05,46,23,5E,23,56,DD,6E,06,DD
380 DATA 66,08,C5,D5,E5,1A,47,CD,06,B9
390 DATA 78,CD,34,9D,47,CD,09,B9,E1,5D
400 DATA 54,CD,75,BB,DD,7E,02,CD,90,BB
410 DATA 78,CD,5A,BB,3C,CD,5A,BB,6B,62
420 DATA 2C,CD,75,BB,DD,7E,00,CD,90,BB
430 DATA 78,3C,3C,CD,5A,BB,3C,CD,5A,BB
440 DATA 6B,62,24,24,D1,13,C1,10,BD,F1
450 DATA CD,90,BB,C9,CD,A5,BB,EB,CD,AE
```

```
460 DATA BB,F5,0E,02,06,04,C5,1A,0F,0F
470 DATA 0F,0F,06,04,1F,CB,1E,CB,2E,10
480 DATA F9,7E,23,77,06,07,23,10,FD,1A
490 DATA 06,04,1F,CB,1E,CB,2E,10,F9,7E
500 DATA 23,77,06,07,2B,10,FD,13,C1,10
510 DATA D3,06,08,23,10,FD,0D,20,C9,F1
520 DATA C9,DD,6E,02,DD,66,06,2D,25,DD
530 DATA 7E,04,94,4F,DD,7E,00,95,57,CD
540 DATA 1A,BC,3E,00,81,10,FD,5F,18,38
550 DATA D5,43,5D,7C,C6,38,57,7D,C6,50
560 DATA 6F,30,0A,24,7C,E6,07,20,04,7C
570 DATA D6,08,67,E5,7E,12,1C,20,0A,14
580 DATA 7A,E6,07,20,04,7A,D6,08,57,2C
590 DATA 20,0A,24,7C,E6,07,20,04,7C,D6
600 DATA 08,67,10,E2,E1,D1,D5,E5,4B,E5
610 DATA 06,07,5D,54,7C,C6,08,67,7E,12
620 DATA 10,F6,E1,2C,20,0A,24,7C,E6,07
630 DATA 20,04,7C,D6,08,67,0D,20,E2,E1
640 DATA D1,15,20,A2,7C,C6,38,67,43,36
650 DATA 00,2C,20,0A,24,7C,E6,07,20,04
660 DATA 7C,D6,08,67,10,EF,C9,DD,6E,00
670 DATA DD,66,06,2D,25,DD,7E,04,94,4F
680 DATA 7D,C6,02,DD,96,02,57,CD,1A,BC
690 DATA 7C,C6,38,67,3E,00,81,10,FD,5F
700 DATA 18,38,D5,43,5D,7C,D6,38,57,7D
710 DATA D6,50,6F,30,0A,7C,25,E6,07,20
```

```
720 DATA 04,7C,C6,08,67,E5,7E,12,1C,20
730 DATA 0A,14,7A,E6,07,20,04,7A,D6,08
740 DATA 57,2C,20,0A,24,7C,E6,07,20,04
750 DATA 7C,D6,08,67,10,E2,E1,D1,D5,E5
760 DATA 4B,E5,06,07,5D,54,7C,D6,08,67
770 DATA 7E,12,10,F6,E1,2C,20,0A,24,7C
780 DATA E6,07,20,04,7C,D6,08,67,0D,20
790 DATA E2,E1,D1,15,20,A2,7C,D6,38,67
800 DATA 43,36,00,2C,20,0A,24,7C,E6,07
810 DATA 20,04,7C,D6,08,67,10,EF,C9,DD
820 DATA 6E,02,DD,66,06,2D,25,DD,7E,04
830 DATA 94,4F,DD,7E,00,95,57,CD,1A,BC
840 DATA 3E,00,81,10,FD,47,CD,11,BC,DA
850 DATA 5B,9F,28,48,DD,6E,02,DD,66,04
860 DATA 2D,25,C5,CD,1A,BC,C1,0E,08,C5
870 DATA E5,B7,CB,16,F5,7D,2D,B7,20,0A
880 DATA 7C,25,E6,07,20,04,7C,C6,08,67
890 DATA F1,10,EB,E1,7C,C6,08,67,C1,0D
900 DATA 20,DF,7C,D6,40,67,7D,C6,50,6F
910 DATA 30,0A,24,7C,E6,07,20,04,7C,D6
920 DATA 08,67,15,20,C6,C9,05,D5,0E,08
930 DATA C5,E5,CB,26,5D,54,2C,20,0A,24
940 DATA 7C,E6,07,20,04,7C,D6,08,67,1A
950 DATA CB,26,38,04,CB,A7,18,02,CB,E7
960 DATA CB,66,28,02,CB,C7,12,10,DD,CB
970 DATA A6,E1,7C,C6,08,67,C1,0D,20,CE
```

```
980 DATA 7C,D6,40,67,7D,C6,50,6F,30,0A
990 DATA 24,7C,E6,07,20,04,7C,D6,08,67
1000 DATA D1,15,20,B3,C9,05,D5,0E,08,C5
1010 DATA E5,C5,5D,54,2C,20,0A,24,7C,E6
1020 DATA 07,20,04,7C,D6,08,67,7E,1F,4F
1030 DATA 1A,17,17,06,04,17,CB,21,CB,21
1040 DATA 17,10,F8,12,C1,10,DC,7E,17,06
1050 DATA 04,07,CB,27,10,FB,77,E1,7C,C6
1060 DATA 08,67,C1,0D,20,C7,7C,D6,40,67
1070 DATA 7D,C6,50,6F,30,0A,24,7C,E6,07
1080 DATA 20,04,7C,D6,08,67,D1,15,20,AC
1090 DATA C9,DD,6E,02,DD,66,04,2D,25,7C
1100 DATA C6,02,DD,96,06,4F,DD,7E,00,95
1110 DATA 57,CD,1A,BC,2B,3E,00,81,23,10
1120 DATA FC,47,CD,11,BC,DA,6F,A0,28,46
1130 DATA DD,6E,02,DD,66,06,2D,25,C5,CD
1140 DATA 1A,BC,C1,0E,08,C5,E5,B7,CB,1E
1150 DATA F5,2C,20,0A,24,7C,E6,07,20,04
1160 DATA 7C,D6,08,67,F1,10,ED,E1,7C,C6
1170 DATA 08,67,C1,0D,20,E1,7C,D6,40,67
1180 DATA 7D,C6,50,6F,30,0A,24,7C,E6,07
1190 DATA 20,04,7C,D6,08,67,15,20,C8,C9
1200 DATA 05,D5,0E,08,C5,E5,CB,3E,5D,54
1210 DATA 7D,2D,B7,20,0A,7C,25,E6,07,20
1220 DATA 04,7C,C6,08,67,1A,CB,3E,38,04
1230 DATA CB,9F,18,02,CB,DF,CB,5E,28,02
1240 DATA CB,FF,12,10,DB,CB,9E,E1,7C,C6
```

```
1250 DATA 08,67,C1,0D,20,CC,7C,D6,40,67
1260 DATA 7D,C6,50,6F,30,0A,24,7C,E6,07
1270 DATA 20,04,7C,D6,08,67,D1,15,20,B1
1280 DATA C9,05,D5,0E,08,C5,E5,C5,5D,54
1290 DATA 7D,2D,B7,20,0A,7C,25,E6,07,20
1300 DATA 04,7C,C6,08,67,7E,17,4F,1A,1F
1310 DATA 1F,06,04,1F,CB,39,CB,39,1F,10
1320 DATA F8,12,C1,10,DA,7E,1F,06,04,0F
1330 DATA CB,3F,10,FB,77,E1,7C,C6,08,67
1340 DATA C1,0D,20,C5,7C,D6,40,67,7D,C6
1350 DATA 50,6F,30,0A,24,7C,E6,07,20,04
1360 DATA 7C,D6,08,67,D1,15,20,AA,C9,21
1370 DATA 68,9C,3E,0D,CD,5A,BB,3E,0A,CD
1380 DATA 5A,BB,3E,7C,CD,5A,BB,7E,23,CB
1390 DATA 7F,20,05,CD,5A,BB,18,F5,CB,BF
1400 DATA CD,5A,BB,3E,0D,CD,5A,BB,3E,0A
1410 DATA CD,5A,BB,7E,B7,20,DD,C9
```

City Bomber

To demonstrate the package, we have the game City Bomber. It is, in our totally unbiased opinion, one of the best versions of the popular "bomb the buildings to make a landing runway" games that we've seen on the Amstrad. The program features a large eight-character UDG-designed plane, a big graphical bomb and good sound effects.

The program shows what can be achieved in a relatively short time using the games writing package.

```
10 SYMBOL AFTER 256:MEMORY 39999:SYMBOL
AFTER 236
20 LOAD "",40000
30 CALL 40000
40 ENT 1,230,1,3,230,1,3
50 ENT -2,6,2,2,4,-2,2
60 RANDOMIZE TIME
70 SYMBOL 240,120,124,62,63,31,31,63,62
80 SYMBOL 241,0,0,0,0,255,255,255,253
90 SYMBOL 242,0,0,0,0,224,24,206,255
100 SYMBOL 243,31,60,112,0,0,0,0,0
110 SYMBOL 244,251,127,15,31,31,62,60,0
120 SYMBOL 245,251,255,240,192,128,0,0,0
130 SYMBOL 246,254,0,0,0,0,0,0,0
140 SYMBOL 247,231,90,231,129,66,36,60,1
26
150 SYMBOL 248,126,255,255,255,255,126,1
26,60
160 SYMBOL 249,255,255,241,241,255,241,2
41,255
170 SYMBOL 250,255,255,143,143,255,143,1
43,255
180 s$="    ":' 4 spaces
190 a$=CHR$(240)+CHR$(241)+CHR$(241)+CHR
$(242)
200 b$=CHR$(243)+CHR$(244)+CHR$(245)+CHR
$(246)
```

```
210 INK 0,14:INK 1,0:INK 2,6:INK 3,24
220 BORDER 14:MODE 1
230 m$="City Bomber":|BIGPRINT,10,1,@m$,
3,2
240 FOR n=5 TO 37 STEP 2
250 x=15-INT(RND*10)
260 PEN 2+n/2 MOD 2
270 FOR y=x TO 25
280 LOCATE n,y:PRINT CHR$(249);CHR$(250)
290 NEXT:NEXT
300 ch%=0:cr%=0:PEN 1:LOCATE 1,1
310 PRINT a$:PRINT b$
320 SOUND 129,500,2000,3,0,0,1
330 v=1:w=1:c=1
340 IF w<23 AND INKEY(47)=0 GOTO 470
350 IF v=30 AND w=23 GOTO 1120
360 FOR n=1 TO 3:NEXT
370 IF c=1 THEN |READCHAR,v+4,w+1,@ch%:I
F ch%>248 GOTO 920
380 |RSCROLL,v,v+4,w,w+1
390 c=c+1:IF c=9 THEN c=1:v=v+1
400 IF v<37 GOTO 340
410 LOCATE 37,w:PRINT s$
420 LOCATE 37,w+1:PRINT s$
430 LOCATE 1,w+2:PRINT a$
440 LOCATE 1,w+3:PRINT b$
450 SOUND 129,500,2000,3,0,0,1
```

```
460 v=1:w=w+2:GOTO 340
470 d=c:x=v+2:y=w+2
480 |READCHAR,x,y,@ch%:IF ch%>32 THEN t=
y:GOTO 710
490 |READCHAR,x,y+1,@ch%:|READCHAR,x,y+2
,@cr%
500 SOUND 130,30,500,5,0,1
510 LOCATE x,y:PRINT CHR$(247):LOCATE x,
y+1:PRINT CHR$(248)
520 IF ch%+cr%>64 GOTO 700
530 IF c=1 THEN |READCHAR,v+4,w+1,@ch%:I
F ch%>248 GOTO 920
540 |RSCROLL,v,v+4,w,w+1
550 |DSCROLL,x,x,y,y+2:|DSCROLL,x,x,y,y+
2
560 c=c+1:IF c=9 THEN c=1:v=v+1
570 IF v<37 GOTO 640
580 LOCATE 37,w:PRINT s$
590 LOCATE 37,w+1:PRINT s$
600 LOCATE 1,w+2:PRINT a$
610 LOCATE 1,w+3:PRINT b$
620 SOUND 129,500,2000,3,0,0,1
630 v=1:w=w+2
640 b=y
650 y=y-(d=c)-(d-1=(c+3) MOD 8)
660 IF b<y THEN |READCHAR,x,y+2,@ch%:IF
ch%>32 GOTO 700
```

```
670 IF y<24 GOTO 530
680 LOCATE x,24:PRINT" ":LOCATE x,25:PRI
NT" "
690 SOUND 130,0,0,0:GOTO 340
700 t=y+2:LOCATE x,y:PRINT" ":LOCATE x+(
x MOD 2=0),y+1:PRINT"  ":' 2 spaces
710 IF t<11 OR t<23 AND RND<0.7 THEN b=t
+INT((24-t)/3) ELSE b=25
720 IF RND>0.6 OR b=25 AND RND>0.3 THEN
q=4 ELSE q=8
730 m=(1+b-t)*q:z=x+1
740 IF x MOD 2=0 THEN z=x:x=x-1
750 SOUND 130,100*q,800,15,0,2
760 FOR n=1 TO m
770 ¦DSCROLL,x,z,t,b
780 IF c=1 THEN ¦READCHAR,v+4,w+1,@ch%:I
F ch%>248 GOTO 920
790 ¦RSCROLL,v,v+4,w,w+1
800 IF q=4 THEN ¦DSCROLL,x,z,t,b
810 c=c+1:IF c=9 THEN c=1:v=v+1
820 IF v<37 GOTO 890
830 LOCATE 37,w:PRINT s$
840 LOCATE 37,w+1:PRINT s$
850 LOCATE 1,w+2:PRINT a$
860 LOCATE 1,w+3:PRINT b$
870 SOUND 129,500,2000,3,0,0,1
880 v=1:w=w+2
```

```
890 NEXT
900 SOUND 130,0,0,0
910 GOTO 340
920 SOUND 135,0,0,0
930 |EXPLODE:INK 2,6,24:INK 3,24,6:SPEED
 INK 2,2
940 IF w>23 GOTO 1050
950 t=w:b=t+2
960 |READCHAR,v,b,@ch%:IF ch%>32 GOTO 98
0
970 b=b+1:IF b<26 GOTO 960
980 b=b-1:m=(b-t-1)*8
990 FOR n=1 TO m:|DSCROLL,v,v+1,t,b:NEXT
1000 t=w:b=t+2
1010 |READCHAR,v+2,b,@ch%:IF ch%>32 GOTO
 1030
1020 b=b+1:IF b<26 GOTO 1010
1030 b=b-1:m=(b-t-1)*8
1040 FOR n=1 TO m:|DSCROLL,v+2,v+3,t,b:N
EXT
1050 PEN 1:FOR m=1 TO 2000:NEXT
1060 LOCATE 12,2:PRINT"Play again? (Y/N)
"
1070 BORDER 1
1080 r$=UPPER$(INKEY$)
1090 IF r$="Y" GOTO 210
```

```
1100 IF r$<>"N" GOTO 1080
1110 END
1120 FOR n=1 TO 88
1130 SOUND 135,100-n,2,7
1140 |RSCROLL,30,40,19,24
1150 IF n MOD 3=0 THEN |USCROLL,30,40,19
,24
1160 NEXT
1170 PEN 2
1180 m$="CONGRATULATIONS":|BIGPRINT,6,12
,@m$,2,3:GOTO 1050
```

The spacebar drops the bombs for you and you must clear the screen of buildings before you can land and take off again. When playing the game, note the smooth scroll across the screen of the aircraft and see if you can see where the new commands have been used.

Appendices

A - Memory Map
B - Z8Ø Op Codes
C - Hexadecimal To Decimal Conversion

Appendix A

Memory Map

There are several points concerning the memory map which need to be mentioned.

Firstly, it is complicated because in the 64K memory available, 64K RAM and 32K of ROM operate - hence the bank switching. Secondly, all the addresses given on the screen map are in hex. Third, notice the six-digit address at the top of the screen map. Ø1ØØØØ is the upper boundary and is, in fact, one above the highest actual address, which is FFFF.

Ø1ØØØØ	DEFAULT SCREEN MEMORY	**Ø1ØØØØ**	UPPER ROM's (bank switched)
CØØØ	STACK, FIRMWARE DATA AND JUMPBLOCK	**CØØØ**	
B1ØØ	FOREGROUND DATA		
ACØØ	BACKGROUND DATA		
????			
	MEMORY POOL		
????	BACKGROUND DATA		
????	FOREGROUND DATA		
ØØ4Ø		**4ØØØ**	LOWER ROM
ØØØØ	FIRMWARE AREA	**ØØØØ**	

Appendix B

Z8Ø Op Codes

Here is a complete list of Z8Ø Op Codes and instructions.

SUMMARY OF Z80 INSTRUCTION FUNCTIONS

```
ADC   A,operand1    ;Add operand and carry to accumulator
ADC   HL,regpair    ;Add register-pair and carry to accumulator
ADD   A,operand1    ;Add operand to accumulator
ADD   HL,regpair    ;Add register-pair to accumulator
ADD   IX,regpair    ;Add register-pair to IX
ADD   IX,IX         ;Add IX to IX
ADD   IY,regpair    ;Add register-pair to IY
ADD   IY,IY         ;Add IY to IY
AND   operand1      ;Logically AND operand to accumulator
BIT   b,operand     ;Test bit b of operand
CALL  addr          ;Call address
CALL  cc,addr       ;Call address if cc is valid
```

```
CCF             ;Complement carry flag

CP    operand1  ;Compare operand to accumulator

CPD             ;Compare (HL) to accumulator, decrement HL
                ;And BC

CPDR            ;Compare (HL) to accumulator, decrement HL
                ;And BC, repeat until BC = 0 or match found

CPI             ;Compare (HL) to accumulator, increment HL
                ;And decrement BC

CPIR            ;Compare (HL) to accumulator, increment HL
                ;And decrement BC, repeat until BC = 0 or
                ;Match found

CPL             ;Complement (invert) accumulator

DAA             ;Decimal adjust accumulator

DEC   operand   ;Decrement operand

DEC   IX        ;Decrement IX

DEC   IY        ;Decrement IY

DEC   SP        ;Decrement stack pointer

DI              ;Disable interrupts

DJNZ  dis       ;Decrement B, jump to displacement if B = 0

EI              ;Enable interrupts .
```

```
EX      (SP),HL     ;Exchange (SP) and HL

EX      (SP),IX     ;Exchange (SP) and IX

EX      (SP),IY     ;Exchange (SP) and IY

EX      AF,AF'      ;Exchange AF with alternate AF

EX      DE,HL       ;Exchange DE and HL

EXX                 ;Exchange general-purpose registers with
                    ;Alternates

HALT                ;Suspect CPU operation

IM      mode        ;Activate interrupt mode

IN      A,(data)    ;Input into accumulator port data

IN      reg,(C)     ;Input into register port C

INC     operand     ;Increment operand

INC     regpair     ;Increment register-pair

INC     IX          ;Increment IX

INC     IY          ;Increment IY

IND                 ;Load (HL) with input from port C,
                    ;Decrement HL and B

INDR                ;Load (HL) with input from port C,
                    ;Decrement HL and B, repeat until B = 0
```

```
INI             ;Load (HL) with input from port C,
                ;Decrement B and increment HL

INIR            ;Load (HL) with input from port C,
                ;Decrement B and increment HL, repeat until
                ;B = 0

JP    (HL)      ;Jump to address of HL

JP    (IX)      ;Jump to address of IX

JP    (IY)      ;Jump to address of IY

JP    addr      ;Jump address

JP    cc,address ;Load program counter with address
                ;If cc is valid

JR    dis       ;Jump displacement

JR    c,dis     ;Jump displacement if c valid

LD    A,I       ;Load interrupt vector into accumulator

LD    A,operand1 ;Load operand into accumulator

LD    A,R       ;Load refresh into accumulator

LD    (BC),A    ;Load accumulator into (BC)

LD    (DE),A    ;Load accumulator into (DE)

LD    (HL),data ;Load data into (HL)

LD    HL,(addr) ;Load (address) into HL
```

```
LD      (HL),reg        ;Load register into (HL)
LD      I,A             ;Load accumulator into interrupt vector
LD      IX,addr         ;Load address into IX
LD      IX,(addr)       ;Load (address) into IX
LD      (IX + dis),data ;Load into IX + displacement data
LD      IY,addr         ;Load address into IY
LD      IY,(addr)       ;Add (address) into IY
LD      (IY + dis),data ;Load into IY + displacement data
LD      (addr),A        ;Load accumulator into (address)
LD      (addr),regpair  ;Load register-pair into (address)
LD      (addr),IX       ;Load IX into (address)
LD      (addr),IY       ;Load IY into (address)
LD      regpair,addr    ;Load address into register-pair
LD      R,A             ;Load accumulator into refresh
LD      reg,operand1    ;Load operand into register
LD      SP,HL           ;Load HL into stack pointer
LD      SP,IX           ;Load IX into stack pointer
LD      SP,IY           ;Load IY into stack pointer
```

```
LDD             ;Load (HL) into (DE), decrement DE, BC and HL

LDDR            ;Load (HL) into (DE), decrement DE, BC and HL,
                ;Repeat until BC = 0

LDI             ;Load (HL) into (DE), increment DE and HL,
                ;Decrement BC

LDIR            ;Load (HL) into (DE), increment DE and HL,
                ;Decrement BC

NEG             ;Negate accumulator

NOP             ;No operation

OR    operand1  ;Logically OR operand to accumulator

OTDR            ;Load (HL) into port C, decrement B and HL,
                ;Repeat until B = 0

OTIR            ;Load (HL) into port C, increment HL,
                ;Decrement B, repeat until B = 0

OUT   (C),reg   ;Load register into port C

OUT   (data),A  ;Load accumulator into port data

OUTD            ;Load (HL) into port C, decrement HL and B

OUTI            ;Load (HL) into port C, increment HL and
                ;Decrement B

POP   regpair   ;Recover register-pair from stack

POP   IX        ;Recover IX from stack
```

```
POP     IY              ;Recover IY from stack

PUSH    regpair         ;Put register-pair on stack

PUSH
IX                      ;Put IX on stack

PUSH
IY                      ;Put IY on stack

RES     b,operand       ;Reset bit b of operand

RET                     ;Return

RET     cc              ;Return if condition valid

RETI                    ;Return from interrupt

RETN                    ;Return from non maskable interrupt

RL      operand         ;Rotate operand left through carry

RLA                     ;Rotate accumulator left through carry

RLC     operand         ;Rotate operand left circular

RLCA                    ;Rotate accumulator left circular

RLD                     ;Rotate digit left and right between (HL)
                        ;And accumulator

RR      operand         ;Rotate operand right through carry

RRA                     ;Rotate accumulator right through carry
```

```
RRC     operand     ;Rotate operand right circular

RRCA                ;Rotate accumulator right circular

RRD                 ;Rotate digit right and left between (HL)
                    ;And accumulator

RST     vector      ;Restart at location vector

SBC     A,operand   ;Subtract operand from accumulator with carry

SBC     HL,repair   ;Subtract register-pair from HL with carry

SCF                 ;Set carry flag

SET     b,operand   ;Set bit b of operand

SLA     operand     ;Shift operand left arithmetic

SRA     operand     ;Shift operand right arithmetic

SRL     operand     ;Shift operand right logical

SUB     operand     ;Subtract operand from accumulator

XOR     operand     ;Exclusive OR operand to accumulator
```

SYMBOLS

```
reg         A, B, C, D, E, H, L.

regpair     HL, BC, DE, SP
```

mode	1, 2, 3
operand1	A, B, C, D, E, H, L, (HL), (IX + dis), (IY + dis), data
operand	A, B, C, D, E, H, L, (HL), (IX + dis), (IY + dis)
dis	displacement offset, a number between 0-255
data	number between 0-255
addr	address between 0-65535
cc	conditions Z, NZ, C, NC, P, M, PE, PO
c	conditions Z, NZ, C, NC
vector	addresses 0, 8, 16, 24, 32, 40, 48 and 56

ALPHABETICAL LIST OF Z80 INSTRUCTIONS

MNEMONIC	OBJECT CODE	FLAGS
ADC A,(HL)	8E	C Z P/V S
ADC A,(IX+Ø)	DD8EØØ	C Z P/V S
ADC A,(IY+Ø)	FD8EØØ	C Z P/V S
ADC A,Ø	CEØØ	C Z P/V S

ADC	A,A	8F	C Z P/V S
ADC	A,B	88	C Z P/V S
ADC	A,C	89	C Z P/V S
ADC	A,D	8A	C Z P/V S
ADC	A,E	8B	C Z P/V S
ADC	A,H	8C	C Z P/V S
ADC	HL,BC	ED4A	C Z P/V S
ADC	HL,DE	ED5A	C Z P/V S
ADC	HL,HL	ED6A	C Z P/V S
ADC	HL,SP	ED7A	C Z P/V S
ADD	A,(HL)	86	C Z P/V S
ADD	A,(IX+0)	DD8600	C Z P/V S
ADD	A,(IY+0)	FD8600	C Z P/V S
ADD	A,0	C600	C Z P/V S
ADD	A,A	87	C Z P/V S
ADD	A,B	80	C Z P/V S
ADD	A,C	81	C Z P/V S
ADD	A,D	82	C Z P/V S

ADD	A,E	83	C Z P/V S
ADD	A,H	84	C Z P/V S
ADD	HL,BC	09	C
ADD	HL,DE	19	C
ADD	HL,HL	29	C
ADD	HL,SP	39	C
ADD	IX,BC	DD09	C
ADD	IX,DE	DD19	C
ADD	IX,IX	DD29	C
ADD	IX,SP	DD39	C
ADD	IY,BC	FD09	C
ADD	IY,DE	FD19	C
ADD	IY,IY	FD29	C
ADD	IY,SP	FD39	C
AND	(HL)	A6	Z P/V S
AND	(IX+0)	DDA600	Z P/V S
AND	(IY+0)	FDA600	Z P/V S
AND	0	E600	Z P/V S

AND	A	A7	Z P/V S
AND	B	A0	Z P/V S
AND	C	A1	Z P/V S
AND	D	A2	Z P/V S
AND	E	A3	Z P/V S
AND	H	A4	Z P/V S
AND	L	A5	Z P/V S
BIT	0,(HL)	CB46	Z
BIT	0,(IX+0)	DDCB0046	Z
BIT	0,(IY+0)	FDCB0046	Z
BIT	0,A	CB47	Z
BIT	0,B	CB40	Z
BIT	0,C	CB41	Z
BIT	0,D	CB42	Z
BIT	0,E	CB43	Z
BIT	0,H	CB44	Z
BIT	0,L	CB45	Z
BIT	1,(HL)	CB4E	Z

BIT	1,(IX+Ø)	DDCBØØ4E	Z
BIT	1,(IY+Ø)	FDCBØØ4E	Z
BIT	1,A	CB4F	Z
BIT	1,B	CB48	Z
BIT	1,C	CB49	Z
BIT	1,D	CB4A	Z
BIT	1,E	CB4B	Z
BIT	1,H	CB4C	Z
BIT	1,L	CB4D	Z
BIT	2,(HL)	CB56	Z
BIT	2,(IX+Ø)	DDCBØØ56	Z
BIT	2,(IY+Ø)	FDCBØØ56	Z
BIT	2,A	CB57	Z
BIT	2,B	CB50	Z
BIT	2,C	CB51	Z
BIT	2,D	CB52	Z
BIT	2,E	CB53	Z
BIT	2,H	CB54	Z

BIT	2,L	CB55	Z
BIT	3,(HL)	CB5E	Z
BIT	3,(IX+0)	DDCB005E	Z
BIT	3,(IY+0)	FDCB005E	Z
BIT	3,A	CB5F	Z
BIT	3,B	CB58	Z
BIT	3,C	CB59	Z
BIT	3,D	CB5A	Z
BIT	3,E	CB5B	Z
BIT	3,H	CB5C	Z
BIT	3,L	CB5D	Z
BIT	4,(HL)	CB66	Z
BIT	4,(IX+0)	DDCB0066	Z
BIT	4,A	CB67	Z
BIT	4,B	CB60	Z
BIT	4,C	CB61	Z
BIT	4,D	CB62	Z
BIT	4,E	CB63	Z

BIT	4,H	CB64	Z
BIT	4,L	CB65	Z
BIT	5,(HL)	CB6E	Z
BIT	5,(IX+Ø)	DDCBØØ6E	Z
BIT	5,(IY+Ø)	FDCBØØ6E	Z
BIT	5,A	CB6F	Z
BIT	5,B	CB68	Z
BIT	5,C	CB69	Z
BIT	5,D	CB6A	Z
BIT	5,E	CB6B	Z
BIT	5,H	CB6C	Z
BIT	5,L	CB6D	Z
BIT	6,(HL)	CB76	Z
BIT	6,(IX+Ø)	DDCBØØ76	Z
BIT	6,(IY+Ø)	DDCBØØ76	Z
BIT	6,A	CB77	Z
BIT	6,B	CB7Ø	Z
BIT	6,D	CB72	Z

BIT 6,E	CB73	Z
BIT 6,H	CB74	Z
BIT 6,L	CB75	Z
BIT 7,(HL)	CB7E	Z
BIT 7,(IX+0)	DDCB007E	Z
BIT 7,(IY+0)	FDCB007E	Z
BIT 7,A	CB7F	Z
BIT 7,B	CB78	Z
BIT 7,C	CB79	Z
BIT 7,D	CB7A	Z
BIT 7,E	CB7B	Z
BIT 7,H	CB7C	Z
BIT 7,L	CB7D	Z
CALL 0	CD0000	
CALL C,0	DC0000	
CALL M,0	FC0000	
CALL NC,0	D40000	
CALL NZ,0	C40000	

CALL	P,Ø	F4ØØØØ	
CALL	PE,Ø	ECØØØØ	
CALL	PO,Ø	E4ØØØØ	
CALL	Z,Ø	CCØØØØ	
CCF		3F	C
CP	Ø	FEØØ	C Z P/V S
CP	(HL)	BE	C Z P/V S
CP	(IX+Ø)	DDBEØØ	C Z P/V S
CP	(IY+Ø)	FDBEØØ	C Z P/V S
CP	A	BF	C Z P/V S
CP	B	B8	C Z P/V S
CP	C	B9	C Z P/V S
CP	D	BA	C Z P/V S
CP	E	BB	C Z P/V S
CP	H	BC	C Z P/V S
CP	L	BD	C Z P/V S
CPD		EDA9	Z P/V S
CPDR		EDB9	Z P/V S

CPI		EDA1	Z P/V S
CPIR		EDB1	Z P/V S
CPL		2F	
DAA		27	C Z P/V S
DEC	(HL)	35	Z P/V S
DEC	(IX+0)	DD3500	Z P/V S
DEC	(IY+0)	FD3500	Z P/V S
DEC	A	3D	Z P/V S
DEC	B	05	Z P/V S
DEC	BC	0B	
DEC	C	0D	Z P/V S
DEC	D	15	Z P/V S
DEC	DE	1B	
DEC	E	1D	Z P/V S
DEC	H	25	Z P/V S
DEC	HL	2B	
DEC	IX	DD2B	
DEC	IY	FD2B	

DEC	L	2D	Z P/V S
DEC	SP	3B	
DI		F3	
DJNZ	Ø	1ØØØ	
EI		FB	
EX	(SP),HL	E3	
EX	(SP),IX	DDE3	
EX	(SP),IY	FDE3	
EX	AF,AF'	Ø8	C Z P/V S
EX	DE,HL	EB	
EXX		D9	
HALT		76	
IM	Ø	ED46	
IM	1	ED56	
IM	2	ED5E	
IN	A,(Ø)	DBØØ	
IN	A,(C)	ED78	Z P/V S
IN	B,(C)	ED4Ø	Z P/V S

IN	C,(C)	ED48	Z P/V S
IN	D,(C)	ED50	Z P/V S
IN	E,(C)	ED58	Z P/V S
IN	H,(C)	ED60	Z P/V S
IN	L,(C)	ED68	Z P/V S
INC	(HL)	34	Z P/V S
INC	(IX+0)	DD3400	Z P/V S
INC	(IY+0)	FD3400	Z P/V S
INC	A	3C	Z P/V S
INC	B	04	Z P/V S
INC	BC	03	
INC	C	0C	Z P/V S
INC	D	14	Z P/V S
INC	DE	13	
INC	E	1C	Z P/V S
INC	H	24	Z P/V S
INC	HL	23	Z P/V S
INC	IX	DD23	

INC	IY	FD23	
INC	L	2C	Z P/V S
INC	SP	33	
IND		EDAA	Z
INDR		EDBA	
INI		EDA2	Z
INIR		EDB2	
JP	Ø	C30000	
JP	(HL)	E9	
JP	(IX)	DDE9	
JP	(IY)	FDE9	
JP	C,Ø	DA0000	
JP	M,Ø	FA0000	
JP	NC,Ø	D20000	
JP	NZ,Ø	C20000	
JP	P,Ø	F20000	
JP	PE,Ø	EA0000	
JP	PO,Ø	E20000	

JP	Z,Ø	CAØØØØ
JR	Ø	18ØØ
JR	C,Ø	38ØØ
JR	NC,Ø	3ØØØ
JR	NZ,Ø	2ØØØ
JR	Z,Ø	28ØØ
LD	(BC),A	Ø2
LD	(DE),A	12
LD	(HL),Ø	36ØØ
LD	(HL),A	77
LD	(HL),B	7Ø
LD	(HL),C	71
LD	(HL),D	72
LD	(HL),E	73
LD	(HL),H	74
LD	(HL),L	75
LD	(IX+Ø),Ø	DD36ØØØØ
LD	(IX+Ø),A	DD77ØØ

LD	(IX+0),B	DD7000
LD	(IX+0),C	DD7100
LD	(IX+0),D	DD7200
LD	(IX+0),E	DD7300
LD	(IX+0),H	DD7400
LD	(IX+0),L	DD7500
LD	(IY+0),0	FD360000
LD	(IY+0),A	FD7700
LD	(IY+0),B	FD7000
LD	(IY+0),C	FD7100
LD	(IY+0),D	FD7200
LD	(IY+0),E	FD7300
LD	(IY+0),H	FD7400
LD	(IY+0),L	FD7500
LD	(0000),A	320000
LD	(0000),BC	ED430000
LD	(0000),DE	ED530000
LD	(0000),HL	220000

LD	(0000),IX	DD220000	
LD	(0000),IY	FD220000	
LD	(0000),SP	ED730000	
LD	A,(BC)	0A	
LD	A,(DE)	1A	
LD	A,(HL)	7E	
LD	A,(IX+0)	DD7E00	
LD	A,(IY+0)	FD7E00	
LD	A,(0000)	3A0000	
LD	A,0	3E00	
LD	A,A	7F	
LD	A,B	78	
LD	A,C	79	
LD	A,D	7A	
LD	A,H	7C	
LD	A,I	ED57	Z P/V S
LD	A,L	7D	
LD	A,R	ED5F	Z P/V S

LD	B,(HL)	46
LD	B,(IX+0)	DD4600
LD	B,(IY+0)	FD4600
LD	B,0	0600
LD	B,A	47
LD	B,B	40
LD	B,C	41
LD	B,D	42
LD	B,E	43
LD	B,H	44
LD	B,L	45
LD	BC,(0000)	ED4B0000
LD	BC,0000	010000
LD	C,(HL)	4E
LD	C,(IX+0)	DD4E00
LD	C,(IY+0)	FD4E00
LD	C,0	0E00
LD	C,A	4F

LD	C,B	48
LD	C,C	49
LD	C,D	4A
LD	C,E	4B
LD	C,H	4C
LD	C,L	4D
LD	D,(HL)	56
LD	D,(IX+0)	DD5600
LD	D,(IY+0)	FD5600
LD	D,0	1600
LD	D,A	57
LD	D,B	50
LD	D,C	51
LD	D,D	52
LD	D,E	53
LD	D,H	54
LD	D,L	55
LD	DE,(0000)	ED5B0000

LD	DE,0000	110000
LD	E,(HL)	5E
LD	E,(IX+0)	DD5E00
LD	E,(IY+0)	FD5E00
LD	E,0	1E00
LD	E,A	5F
LD	E,B	58
LD	E,C	59
LD	E,D	5A
LD	E,E	5B
LD	E,H	5C
LD	E,L	5D
LD	H,(HL)	66
LD	H,(IX+0)	DD6600
LD	H,(IY+0)	FD6600
LD	H,0	2600
LD	H,A	67
LD	H,B	60

LD	H,C	61
LD	H,D	62
LD	H,E	63
LD	H,H	64
LD	H,L	65
LD	HL,(0000)	2A0000
LD	HL,0000	210000
LD	I,A	ED47
LD	IX,(0000)	DD2A0000
LD	IX,0000	DD210000
LD	IY,(0000)	FD2A0000
LD	IY,0000	FD210000
LD	L,(HL)	6E
LD	L,(IX+0)	DD6E00
LD	L,(IY+0)	FD6E00
LD	L,0	2E00
LD	L,A	6F
LD	L,B	68

LD	L,C	69	
LD	L,D	6A	
LD	L,E	6B	
LD	L,H	6C	
LD	L,L	6D	
LD	R,A	ED4F	
LD	SP,(0000)	ED7B0000	
LD	SP,0000	310000	
LD	SP,HL	F9	
LD	SP,IX	DDF9	
LD	SP,IY	FDF9	
LDD		EDA8	P/V
LDDR		EDB8	
LDI		EDA0	P/V
LDIR		EDB0	
NEG		ED44	C Z P/V S
NOP		00	
OR	(HL)	B6	Z P/V S

OR	(IX+0)	DDB600	Z P/V S
OR	(IY+0)	FDB600	Z P/V S
OR	0	F600	Z P/V S
OR	A	B7	Z P/V S
OR	B	B0	Z P/V S
OR	C	B1	Z P/V S
OR	D	B2	Z P/V S
OR	E	B3	Z P/V S
OR	H	B4	Z P/V S
OR	L	B5	Z P/V S
OTDR		EDBB	
OTIR		EDB3	
OUT	(C),A	ED79	
OUT	(C),B	ED41	
OUT	(C),C	ED49	
OUT	(C),D	ED51	
OUT	(C),E	ED59	
OUT	(C),H	ED61	

OUT (C),L	ED69	
OUT (Ø),A	D3ØØ	
OUTD	EDAB	Z
OUTI	EDA3	Z
POP AF	F1	C Z P/V S
POP BC	C1	
POP DE	D1	
POP HL	E1	
POP IX	DDE1	
POP IY	FDE1	
PUSH AF	F5	
PUSH BC	C5	
PUSH DE	D5	
PUSH HL	E5	
PUSH IX	DDE5	
PUSH IY	FDE5	
RES Ø,(HL)	CB86	
RES Ø,(IX+Ø)	DDCBØØ86	

RES	0,(IY+0)	FDCB0086
RES	0,A	CB87
RES	0,B	CB80
RES	0,C	CB81
RES	0,D	CB82
RES	0,E	CB83
RES	0,H	CB84
RES	0,L	CB85
RES	1,(HL)	CB8E
RES	1,(IX+0)	DDCB008E
RES	1,(IY+0)	FDCB008E
RES	1,A	CB8F
RES	1,B	CB88
RES	1,C	CB89
RES	1,D	CB8A
RES	1,E	CB8B
RES	1,H	CB8C
RES	1,L	CB8D

RES	2,(HL)	CB96
RES	2,(IX+Ø)	DDCBØØ96
RES	2,(IY+Ø)	FDCBØØ96
RES	2,A	CB97
RES	2,B	CB9Ø
RES	2,C	CB91
RES	2,D	CB92
RES	2,E	CB93
RES	2,H	CB94
RES	2,L	CB95
RES	3,(HL)	CB9E
RES	3,(IX+Ø)	DDCBØØ9E
RES	3,(IY+Ø)	FDCBØØ9E
RES	3,A	CB9F
RES	3,B	CB98
RES	3,C	CB99
RES	3,D	CB9A
RES	3,E	CB9B

RES	3,H	CB9C
RES	3,L	CB9D
RES	4,(HL)	CBA6
RES	4,(IX+0)	DDCB00A6
RES	4,(IY+0)	FDCB00A6
RES	4,A	CBA7
RES	4,B	CBA0
RES	4,C	CBA1
RES	4,D	CBA2
RES	4,E	CBA3
RES	4,H	CBA4
RES	4,L	CBA5
RES	5,(HL)	CBAE
RES	5,(IX+0)	DDCB00AE
RES	5,(IY+0)	FDCB00AE
RES	5,A	CBAF
RES	5,B	CBA8
RES	5,C	CBA9

RES	7,L	CBBD	
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		ED4D	
RETN		ED45	
RL	(HL)	CB16	C Z P/V S
RL	(IX+0)	DDCB0016	C Z P/V S
RL	(IY+0)	FDCB0016	C Z P/V S
RL	A	CB17	C Z P/V S
RL	B	CB10	C Z P/V S
RL	C	CB11	C Z P/V S

RL	D	CB12	C Z P/V S
RL	E	CB13	C Z P/V S
RL	H	CB14	C Z P/V S
RL	L	CB15	C Z P/V S
RLA		17	C
RES	5,D	CBAA	
RES	5,E	CBAB	
RES	5,H	CBAC	
RES	5,L	CBAD	
RES	6,(HL)	CBB6	
RES	6,(IX+Ø)	DDCBØØB6	
RES	6,(IY+Ø)	FDCBØØB6	
RES	6,A	CBB7	
RES	6,B	CBBØ	
RES	6,C	CBB1	
RES	6,D	CBB2	
RES	6,E	CBB3	
RES	6,H	CBB4	

RES	6,L	CBB5	
RES	7,(HL)	CBFE	
RES	7,(IX+Ø)	DDCBØØBE	
RES	7,(IY+Ø)	FDCBØØBE	
RES	7,A	CBBF	
RES	7,B	CBB8	
RES	7,C	CBB9	
RES	7,D	CBBA	
RES	7,E	CBBB	
RES	7,H	CBBC	
RLC	(HL)	CBØ6	C Z P/V S
RLC	(IX+Ø)	DDCBØØ16	C Z P/V S
RLC	(IY+Ø)	FDCBØØ16	C Z P/V S
RLC	A	CBØ7	C Z P/V S
RLC	B	CBØØ	C Z P/V S
RLC	C	CBØ1	C Z P/V S
RLC	D	CBØ2	C Z P/V S
RLC	E	CBØ3	C Z P/V S

RLC	H	CB04	C Z P/V S
RLC	L	CB05	C Z P/V S
RLCA		07	C
RLD		ED6F	Z P/V S
RR	(HL)	CB1E	C Z P/V S
RR	(IX+0)	DDCB001E	C Z P/V S
RR	(IY+0)	FDCB001E	C Z P/V S
RR	A	CB1F	C Z P/V S
RR	B	CB18	C Z P/V S
RR	C	CB19	C Z P/V S
RR	D	CB20	C Z P/V S
RR	E	CB21	C Z P/V S
RR	H	CB22	C Z P/V S
RR	L	CB23	C Z P/V S
RRA		1F	C
RRC	(HL)	CB0E	C Z P/V S
RRC	(IX+0)	DDCB000E	C Z P/V S
RRC	(IY+0)	FDCB000E	C Z P/V S

RRC	A	CB0F	C Z P/V S
RRC	B	CB08	C Z P/V S
RRC	C	CB09	C Z P/V S
RRC	D	CB0A	C Z P/V S
RRC	E	CB0B	C Z P/V S
RRC	H	CB0C	C Z P/V S
RRC	L	CB0D	C Z P/V S
RRCA		0F	C
RRD			Z P/V S
RST	00H	C7	
RST	08H	CF	
RST	10H	D7	
RST	18H	DF	
RST	20H	E7	
RST	28H	EF	
RST	30H	F7	
RST	38H	FF	
SBC	A,0	DE00	C Z P/V S

SBC	A,(HL)	9E	C Z P/V S
SBC	A,(IX+0)	DD9E00	C Z P/V S
SBC	A,(IY+0)	FD9E00	C Z P/V S
SBC	A,A	9F	C Z P/V S
SBC	A,B	98	C Z P/V S
SBC	A,C	99	C Z P/V S
SBC	A,D	9A	C Z P/V S
SBC	A,E	9B	C Z P/V S
SBC	A,H	9C	C Z P/V S
SBC	A,L	9D	C Z P/V S
SBC	HL,BC	ED42	C Z P/V S
SBC	HL,DE	ED52	C Z P/V S
SBC	HL,HL	ED62	C Z P/V S
SBC	HL,SP	ED72	C Z P/V S
SCF		37	C
SET	0,(HL)	CBC6	
SET	0,(IX+0)	DDCB00C6	
SET	0,(IY+0)	FDCB00C6	

SET	0,A	CBC7
SET	0,B	CBC0
SET	0,C	CBC1
SET	0,D	CBC2
SET	0,E	CBC3
SET	0,H	CBC4
SET	0,L	CBC5
SET	1,(HL)	CBCE
SET	1,(IX+0)	DDCB00CE
SET	1,(IY+0)	FDCB00CE
SET	1,A	CBCF
SET	1,B	CBC8
SET	1,C	CBC9
SET	1,D	CBCA
SET	1,E	CBCB
SET	1,H	CBCC
SET	1,L	CBCD
SET	2,(HL)	CBD6

SET	2,(IX+0)	DDCB00D6
SET	2,(IY+0)	FDCB00D6
SET	2,A	CBD7
SET	2,B	CBD0
SET	2,C	CBD1
SET	2,D	CBD2
SET	2,E	CBD3
SET	2,H	CBD4
SET	2,L	CBD5
SET	3,(HL)	CBDE
SET	3,(IX+0)	DDCB00DE
SET	3,(IY+0)	FDCB00DE
SET	3,A	CBDF
SET	3,B	CBD8
SET	3,C	CBD9
SET	3,D	CBDA
SET	3,E	CBDB
SET	3,H	CBDC

SET	3,L	CBDD
SET	4,(HL)	CBE6
SET	4,(IX+0)	DDCB00E6
SET	4,(IY+0)	FDCB00E6
SET	4,A	CBE7
SET	4,B	CBE0
SET	4,C	CBE1
SET	4,D	CBE2
SET	4,E	CBE3
SET	4,H	CBE4
SET	4,L	CBE5
SET	5,(HL)	CBEE
SET	5,(IX+0)	DDCB00EE
SET	5,(IY+0)	FDCB00EE
SET	5,A	CBEF
SET	5,B	CBE8
SET	5,C	CBE9
SET	5,D	CBEA

SET	5,H	CBEB
SET	5,L	CBEC
SET	6,(HL)	CBF6
SET	6,(IX+0)	DDCB00F6
SET	6,(IY+0)	FDCB00F6
SET	6,A	CBF7
SET	6,B	CBF0
SET	6,C	CBF1
SET	6,D	CBF2
SET	6,E	CBF3
SET	6,H	CBF4
SET	6,L	CBF5
SET	7,(HL)	CBFE
SET	7,(IX+0)	DDCB00FE
SET	7,(IY+0)	FDCB00FE
SET	7,A	CBFF
SET	7,B	CBF8
SET	7,C	CBF9

SET	7,D	CBFA	
SET	7,E	CBFB	
SET	7,H	CBFC	
SET	7,L	CBFD	
SLA	(HL)	CB26	C Z P/V S
SLA	(IX+0)	DDCB0026	C Z P/V S
SLA	(IY+0)	FDCB0026	C Z P/V S
SLA	A	CB27	C Z P/V S
SLA	B	CB20	C Z P/V S
SLA	C	CB21	C Z P/V S
SLA	D	CB22	C Z P/V S
SLA	E	CB23	C Z P/V S
SLA	H	CB24	C Z P/V S
SLA	L	CB25	C Z P/V S
SRA	(HL)	CB2E	C Z P/V S
SRA	(IX+0)	DDCB002E	C Z P/V S
SRA	(IY+0)	FDCB002E	C Z P/V S
SRA	A	CB2F	C Z P/V S

SRA	B	CB28	C Z P/V S
SRA	C	CB29	C Z P/V S
SRA	D	CB2A	C Z P/V S
SRA	E	CB2B	C Z P/V S
SRA	H	CB2C	C Z P/V S
SRA	L	CB2D	C Z P/V S
SRL	(HL)	CB3E	C Z P/V S
SRL	(IX+0)	DDCB003E	C Z P/V S
SRL	(IY+0)	FDCB003E	C Z P/V S
SRL	A	CB3F	C Z P/V S
SRL	B	CB38	C Z P/V S
SRL	C	CB39	C Z P/V S
SRL	D	CB3A	C Z P/V S
SRL	E	CB3B	C Z P/V S
SRL	H	CB3C	C Z P/V S
SRL	L	CB3D	C Z P/V S
SUB	0	D600	C Z P/V S

SUB	(HL)	96	C Z P/V S
SUB	(IX+Ø)	DD9600	C Z P/V S
SUB	(IY+Ø)	FD9600	C Z P/V S
SUB	A	97	C Z P/V S
SUB	B	90	C Z P/V S
SUB	C	91	C Z P/V S
SUB	D	92	C Z P/V S
SUB	E	93	C Z P/V S
SUB	H	94	C Z P/V S
SUB	L	95	C Z P/V S
XOR	(HL)	AE	Z P/V S
XOR	(IX+Ø)	DDAE00	Z P/V S
XOR	(IY+Ø)	FDAE00	Z P/V S
XOR	Ø	EE00	Z P/V S
XOR	A	AF	Z P/V S
XOR	B	A8	Z P/V S
XOR	C	A9	Z P/V S
XOR	D	AA	Z P/V S

XOR	E	AB	Z P/V S
XOR	H	AC	Z P/V S
XOR	L	AD	Z P/V S

Appendix C

Hexadecimal to Decimal Conversion

Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Now you can really get to grips with machine code on your Amstrad. Written by Clive Gifford and Scott Vincent, two expert programmers who've proved in previous books they can write clear, easy-to-understand English, this book is a certain path to machine code mastery.

The book also includes an invaluable collection of machine code routines (including pixel by pixel scrolling in four directions) which you can incorporate into your BASIC programs, even if you haven't a clue how they work! These 'pre-packaged' machine code routines are designed to make creating arcade and animated games as simple as possible. A complete arcade game is also included in the book to show them in action.

Chapter headings include:

- **Your first machine code program**
- **Passing parameters to and from machine code**
- **Simple arithmetic**
- **Stacking and jumping**
- **Logical operators, and manipulating bits**
- **The Amstrad's screen and ROM routines**
- **The Games Writing Package**
- **Z80 Op Codes**

Designed to work with the 464, 664 and 6128 models, this book will get you into the driver's seat in a matter of hours. You can now command the wizardry of machine code magic for yourself.

ISBN 0-907563-91-0

£8.95

MASTERING MACHINE CODE ON YOUR AMSTRAD 464/664/6128

Clive Gifford
& Scott Vincent

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/