

Amstrad
CPC 464



Melbourne
House

**Amstrad
Machine
Language**
for the
**Absolute
Beginner**

Joe Pritchard



Amstrad
CPC 464
**Machine
Language**
**for the
Absolute
Beginner**

Joe Pritchard



**MELBOURNE HOUSE
PUBLISHERS**

© 1985 Joe Pritchard

All rights reserved. This book is copyright and no part may be copied or stored by electromagnetic, electronic, photographic, mechanical or any other means whatsoever except as provided by national law. All enquiries should be addressed to the publishers:

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

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

Cataloguing in Publication

ISBN 0 86161 193 4

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

I would like to acknowledge the help given by two groups of people in the production of this book; my publisher, for the rapid delivery of a machine and the support given me in the production of the book, and the staff of Amsoft for all their help with my occasional queries.

Finally, I'd like to dedicate this book to my wife, Nicky, who had to put up with it's production!

Joe Pritchard

Contents

CHAPTER 1 - Machine Code First Principles	1
Why Bother?	1
What is Machine Code?	2
The BASIC Interpreter and the OPERATING SYSTEM	2
Disadvantages of Machine Code	3
Assembly Language Programming	3
The Z80 CPU. What can it do?	5
Amstrad Hardware	8
The Z80 CPU	8
The Stack Pointer	11
Memory	12
The CRTG	13
The PSG	13
The PPI	13
The Gate Array	13
CHAPTER 2 - How Computers Count	15
Bits and Bytes	19
Representation of Information	20
Program Representation	21
Data	21
Summing Up	24
CHAPTER 3 - Machine Code meets BASIC	25
Homes for machine code programs	25
POKE and PEEK	29
CALL	30
Saving bytes on tape	32
RUNning a file	32
Loading a file	33
CHAPTER 4 - Registers at work	35
Register Addressing	36
Immediate Addressing	36
Register Indirect Addressing	37
Extended Addressing	39
Labels in Machine Code	42
Indexed Addressing	42
Immediate Indexed Addressing	43
CHAPTER 5 - Passing Parameters to programs	47
Integer variables and Numbers	48
Variables prefixed with @	49
Passing Strings	51
String Descriptor Block	51
CHAPTER 6 - 8 bit counting	55
The F Register	55
Counting with 8 bits	59
8 bit Arithmetic	62
Comparing Numbers	68
Logical Operations	70
Manipulating Bits in a Byte	75
CHAPTER 7 - 16 bit transfers	81
Transfers between register pairs and memory	82
Manipulating the Stack	83

CHAPTER 8	- 16 bit arithmetic and counting	89
	Increment and Decrement	89
	Addition and Subtraction	90
CHAPTER 9	- Loops, Jumps and Block Operations	93
	Jumps	93
	FOR ... NEXT Loops in machine code	98
	CALL and RETURN	101
	Interrupts	105
	Block Operations	106
CHAPTER 10	- Ins and Outs and Odds and Ends	111
	Input and Output Instructions	111
	Odds and Ends	113
CHAPTER 11	- Amstrad Sound	117
	General Notes on the AY-3-8912	117
	PSG Registers	118
	Amplitude Control	121
	Noise	124
	Envelopes	126
CHAPTER 12	- The Amstrad Keyboard	131
	Wait for a key	131
	Don't Wait For a Key	133
CHAPTER 13	- The Amstrad Display	135
	Printing Characters to the Screen	135
	Simple Machine Code Graphics	140
APPENDIX 1	- Instructions and Op-codes	145
APPENDIX 2	- Flag Operation Summary	149
APPENDIX 3	- Numbers on the Amstrad	151
	BIN\$(value,no. of digits)	151
	HEX\$(value)	151
APPENDIX 4	- Timing Programs	153

Chapter 1

Machine Code First Principles

This book is designed to introduce the AMSTRAD BASIC programmer to the "native language" of his computer. This language is called Machine Language, or Machine Code. You may have heard of it before, or it may be a totally new subject to you. Don't worry, the first chapter of this book will slowly and painlessly introduce you to Machine Language ideas and concepts, step by step.

The first thing to do is to look at how we usually program our computer. We type in lines of BASIC, and this instructs the computer to do some task or other. However, we're not actually communicating with the "brain" of the computer when we do this. This brain, called the Central Processor Unit, or CPU, is never spoken to directly while we program the computer in BASIC. We always go through a "middle man", called the BASIC interpreter, when we program our computer in BASIC. However, more about this later.

The CPU used in the Amstrad is called the Z80, and is probably the most popular CPU around in home computers at the moment. There are several other electronic "chips" in the Amstrad, but the CPU is at the heart of all the operations performed by the computer. Indeed, when we talk about programming the Amstrad in Machine Language, we're actually talking about programming the Z80 CPU in Z80 Machine Language.

Why Bother?

Amstrad BASIC, as you're probably found out for yourself, is very powerful. Why should we bother learning a new language? Well, there are three main advantages that using Machine Language offers us over using BASIC. These are:

- (a) Faster Programs
- (b) Programs in Machine Language are more economic in terms of memory.
- (c) Certain tasks can ONLY be done using machine language.

In addition, machine language programming enables us to free ourselves from the restraints of the

Amstrad BASIC Interpreter, and it enables us to alter the way in which the BASIC Interpreter works. I think that you'd agree, therefore, that a knowledge of machine language programming could be rather useful!

Well, having answered the question of Why?, let's look at What machine language is.

What is Machine Code?

The Z80 CPU, if you're never seen one, is a large black chip with 40 "legs" on it. These legs, or pins, are the means by which CPU communicates with the rest of the computer. Of these, there are 8 pins of particular importance which control how the CPU behaves. The CPU communicates with the rest of the computer system by means of electrical signals, and the CPU was designed so as to behave in different ways depending upon the combinations of electrical signals on these 8 pins. Remembering that we're talking about electrical signals, let's represent the presence of a signal by '1' and the absence of a signal on any of these 8 pins as '0'. As there are 8 pins of interest to us at this time, a typical combination of signals might be represented by

01101101

This particular combination of signals will cause the CPU to behave in a particular fashion, and we might say that this combination instructs the CPU to perform a certain job.

We call such a combination of signals, therefore, a Machine Language Instruction, just as LET A=0 is a BASIC Instruction. This is essentially what Machine Code programming is all about; a combination of electrical signals that are capable of causing the CPU to perform a particular task. The instructions that are understood by the Z80 CPU are collectively called the Z80 INSTRUCTION SET. Each different CPU has a different Instruction Set; thus programs written in the machine language of one CPU will almost certainly NOT work properly on another CPU.

The BASIC Interpreter and the OPERATING SYSTEM

The BASIC Interpreter is a machine language program whose job it is to convert the BASIC instructions that are typed in to the computer into Machine Language instructions that the CPU can understand.

The CPU does not understand BASIC, and so the BASIC Instructions must be translated into machine language instructions before the CPU can do anything with them. It's just like the way we might translate from English to French; we'd use a dictionary, or, if we could afford it, the services of a professional Interpreter.

The OPERATING SYSTEM is as machine language program that tells the CPU how to communicate with the monitor, the keyboard and the tape recorder. It's used by the BASIC Interpreter whenever the BASIC instructions need to use any of these devices. Thus the Operating System machine code program for putting a character on the screen will be called whenever a PRINT statement is encountered in BASIC.

The fact that we have to translate BASIC instructions into machine code instructions before we can do anything with them explains why BASIC is slower than machine code programs. The translation process takes time, and often the machine code instructions that are generated by the translation process are not as efficient for the particular job as they might be.

Also, as we've already said, machine language programming gives us the chance to make the computer do things that the Operating System and the BASIC Interpreter never intended us to do.

Disadvantages of Machine Code

There are some disadvantages with writing programs in machine code. Just to set the balance right, I've listed them below:

- (a) Machine Language programs are difficult to read and find errors in.
- (b) They are difficult to transfer on to other computers. Most machine code programs cannot be transferred to other machines without rewriting the programs!
- (c) Needs large numbers of simple instructions in many programs.
- (d) Complex arithmetic is very difficult in machine language.

You can thus see that "You pays your money and you takes your choice" with regard to whether you write a particular program in BASIC or Machine Code. You'd be ill advised, however, to write an accounts package in Machine Code, but it would be equally silly to write a program requiring speed in BASIC.

Assembly Language Programming

Writing machine code programs as strings of 1's and 0's would soon put every one except the hardened professional off the whole idea. So, it's not surprising that there are other ways of representing machine language programs.

The combination of 1's and 0's that represent a machine language instruction can represent many things to the CPU. To us mere humans, they can be seen to represent a BINARY NUMBER. We can thus convert this binary number into a decimal number, but we must

remember that while this is a convenient representation for us, the CPU will still see these numbers as combinations of 1's and 0's on the 8 pins we spoke of at the start of this chapter. We could therefore write a sequence of instructions down as a series of decimal numbers.

While this would be more legible to us than a series of binary numbers, it still doesn't give us any idea of what the instructions actually do. We could, of course, have a list of numbers and the actions that the instructions represented by these numbers perform. What would be useful, of course, would be a method of representing machine language instructions in some form of English. We can, in fact, do this.

We use a form of representation known as ASSEMBLY LANGUAGE. Each machine code instruction is represented by a short, descriptive name called a MNEMONIC. Each mnemonic is also called an ASSEMBLER INSTRUCTION. Thus, using our different ways of representing machine language instructions - binary, decimal and Assembler, we could write a particular instruction down as:

binary	01110110
decimal	118
Assembler	HALT

You might even be able to guess from the mnemonic what this instruction tells the CPU to do. Yes, it tells the CPU to stop, or HALT, until further notice. Assembler Instructions are, as you can probably guess, totally incomprehensible to the CPU. We thus require a means of converting the Assembler Instructions into the machine code instructions and then into sets and electrical signals before the CPU can perform what is expected of it. You can do this conversion yourself, using the tables in the back of this book, or you can use a computer program to do the job for you. Such a program does a similar job to that performed by the BASIC Interpreter, and converts the Assembler Instructions into machine code instructions.

Such a program is called an ASSEMBLER, and it ASSEMBLES the machine code program from the Assembly Language program. If you convert the Assembler instructions into machine code instructions using tables such as those in the back of the book, then the process is called HAND ASSEMBLY. (It's actually quite good practice to start this way).

So far, we've not really seen what machine language can do for us. As a brief interlude, I offer a single instruction machine code program that totally disables ESCAPE and SHIFT-CTRL-ESCAPE from within a running BASIC program. Once this machine code program has been entered, the only way to stop a running program is either hope for an END or an error in the program, or turn the computer off! Not even ON BREAK

offers this degree of protection from a running program being stopped! The instruction simply causes the CPU to ignore any Break events, such as those caused by Escape. Type the line in exactly as shown. Explanations will come later.

```
10 POKE &BDEE,201
```

The rest of your program can now be entered, and once running it will be immune from people pressing the Escape key. You must admit, that single instruction machine code program does something that cannot be done from BASIC. Now we've had a brief glimpse of what machine code can do, let's see what sort of things the CPU can do.

The Z80 CPU. What can it do?

The CPU is responsible for virtually everything that goes on in the computer; as soon as you turn the machine on, the CPU starts running the BASIC Interpreter program and this enables you to type in your programs and commands.

The first thing to realise about the CPU is that it is only able to do simple tasks, such as addition and subtraction, but it can do them very quickly.

Secondly, whereas we might use pencil and paper to do such simple tasks, the CPU doesn't; it performs the tasks in the same way that a child might, that is, with its "fingers". The only use of "pencil and paper" by the CPU is when it is told to store the results of a task. The results are stored in "boxes" within the computer memory.

Two points become obvious from the fact that the CPU uses "fingers" to count on:

- (a) Only whole numbers, or integers, can be dealt with directly by the CPU. (It cannot work in half fingers).
- (b) The numbers involved are obviously limited in size by the number of fingers that the CPU has.
- (c) With respect to the second point above, if the CPU needs to, it can count on its "toes" as well as its fingers!

There is some consolation, however, in that whereas we're stuck with two hands and two feet, the CPU has several more than this. Also, the CPU has 8 "fingers" on each of its hands, and 16 "toes" on each foot. (Note that the CPU can count up to 255 on one of its 8 fingered hands and up to 65535 on one of its feet. Exactly how this is done will be revealed in the next Chapter.)

Let's get back to what the CPU can do. How might it do a simple addition, such as 3+4? Let's write down

the way in which the CPU might do this in mnemonic form. For the sake of this example, let's call one of the CPU hands "A". We'll also allow the CPU to use various "boxes" in memory to store results and other bits of information that are required.

```
LD   A,3
LD   (BOX#1),A   put 3 in box 1
LD   A,4
LD   (BOX#2),A   put 4 in box 2
LD   A,(BOX#1)   get 3 into hand A
ADD  (BOX#2),A   do the sum
LD   (BOX#3),A   store the result
```

LD is the mnemonic for LOAD, and we're simply loading hand "A" in the first instruction with the value 3 - i.e. we're counting up to 3 on the fingers of hand "A". The second instruction where we store this value in a "box", is quite interesting and introduces a rather important concept in machine code programming. The brackets in this instruction indicate that we're interested in the CONTENTS of the box mentioned in the brackets. In this case, the current contents of box #1 are to be replaced by the number that is counted out on the fingers of the "A" hand. This may remind you of the idea of the BASIC variable. However, this box is not the same as a variable - it is simply a location in memory that the programmer has decided to use for a particular purpose in that program. The ADD instruction performs the actual operation, leaving the result on the fingers of the "A" hand. This is then stored away for future use. As you can see, it's a lot more long winded than the BASIC: LET A=3+4

The Stack

Despite the CPU having 8 hands, each with 8 fingers, and two 16-toed feet, it still occasionally finds the need for more places to store numbers. Well, we could use some memory locations, or boxes, as we did above. Sometimes, however, this isn't desirable.

An alternative temporary store that the CPU can use is called the STACK.

For the moment, we'll look at the stack as one of those spikes that well organised people, unlike me, use to put pieces of paper on. Onto the spike paper slips are pushed, and so it's obvious that the last piece of paper pushed on to the spike is the most accessible. In a similar fashion, the most accessible piece of information on the Stack is the last piece of information that was placed on the stack. This is how the Stack is useful to the CPU; it always knows where it put a certain piece of information if it places it on the stack.

The CPU stores information on the Stack or PUSHES

it onto the stack, from one of its hands whenever it needs to use that hand for something else, but still wants easy access to the contents of that hand. Once the CPU wants the information back, it POPS the information back from the stack on to the hand. The CPU can store information from as many of its hands and feet as it likes, each storage operation requiring a separate PUSH. One thing to note about the stack in the computer; the office spike was such that each PUSH caused the stack to increase in height. In the computer, the stack is upside down, and grows downwards as more information is PUSHed onto it.

What the CPU is capable of

As we mentioned earlier in the Chapter, the CPU is really only capable of performing simple tasks. Because the CPU's counting abilities are limited to what it can count up to on its fingers and toes, it is limited to numbers in the following ranges:

- (a) 8 fingered numbers between 0 and 255
- (b) 16 fingered numbers between 0 and 65535

I use the phrase 16-fingered numbers deliberately; we can get the CPU to use two of its 8-fingered hands as an extra "foot" if we so desire. This 2-handed number is thus the same as a number that can be represented on a CPU foot. Now we've met the types of number that the CPU knows how to handle, let's go and examine the instructions that the CPU can understand. They fall into the below main categories.

- (a) Counting on one hand.
- (b) Counting on 2 hands.
- (c) Addition and Subtraction on 1 hand.
- (d) Addition and Subtraction on 2 hands.
- (e) Various manipulations of 1 handed numbers; e.g. Making a number negative.
- (f) Causing the CPU to jump from one point in a machine language program to another.
- (g) Causing the CPU to transfer 8 finger numbers from and to other devices in the computer system.

Before we leave this Chapter of first principles let's have a brief look at two other things that will be useful to us. These are the idea of ADDRESSES, and some details of the Amstrad hardware.

Addresses

You may have noticed the word address turning up occasionally in this Chapter; in normal English, it refers to where a particular house or building can be found in a town full of them. In computing, the address refers to the location within a computer memory where a particular number can be found. This

number could be a machine language instruction or a piece of data, but all the numbers held at the various addresses can be held as 8-fingered numbers - i.e. they are all between 0 and 255 in value. In the Amstrad, some of the memory locations are used for storing the programs that we type in, others store the BASIC Interpreter program, and yet others are used to store the information that is used to form the image on the monitor screen.

Amstrad Hardware

Let's take a brief detour to look at the various electronic components that make up the Amstrad Computer System. Hardware is the term that is applied to the various bits of electronic equipment that make up a computer system. Software is the name given to the programs that we run on the computer. Some wits have suggested that the hardware is the bit you can kick, but I don't consider this to be a useful definition! Figure 2 shows how the various bits of the Amstrad system fit together. We'll now go on to take a brief look at the roles of the devices. As they are all to some extent under the control of the CPU, we'll start by taking a closer look at the CPU.

The Z80 CPU

We've already talked about what it does in general terms; now we'll look at how it's arranged.

There are 8 "hands" in the CPU, and they're all given names. They are called A, B, C, D, E, F, H and L. There are two CPU "feet" called IX and IY. These "hands" and "feet" are often represented in diagrams as

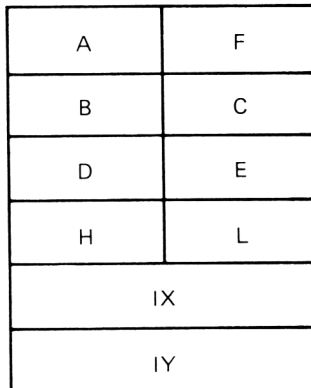


Figure 1. The Z80 Register Set

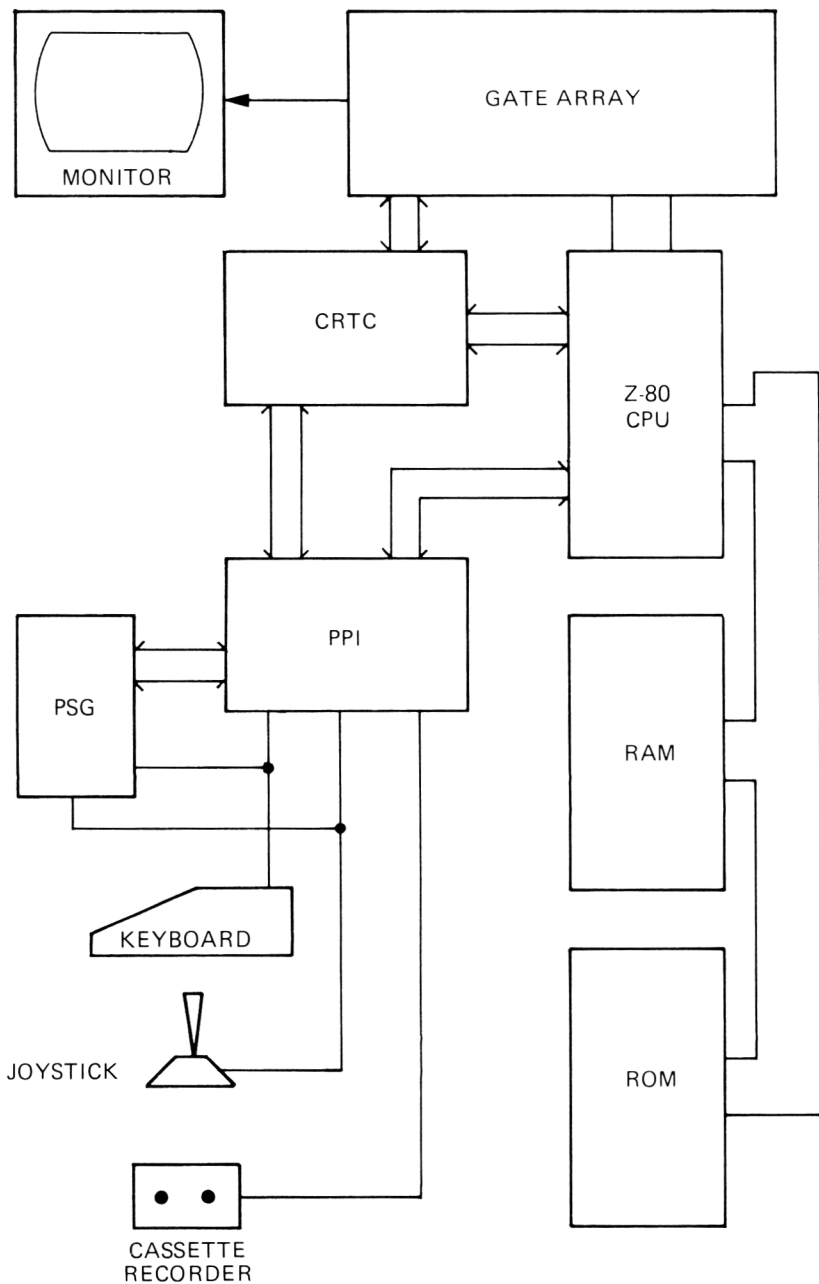


Figure 2. Amstrad Hardware Configuration

All the hands except F can be used for counting on; hand F has a rather special function, as each finger of this hand is used to indicate whether or not a particular event has happened within the CPU. This will be discussed in greater detail when we look at the various instructions that affect the fingers of this hand.

We can, if we want to, team up hands B and C, D and E and H and L to form some new "feet" called BC, DE and HL. Note that a DL foot is not possible, neither is a DH, CE etc. Each of these new feet is capable of holding a 16 finger number just like the IX and IY feet.

The hands and feet of the CPU are usually referred to as CPU REGISTERS. Thus the A hand is usually called the A REGISTER. The feet that we make by pairing up, say, the B and C registers are called REGISTER PAIRS. Thus you can have the BC, DE and HL Register Pairs. The IX and IY feet are also given special names; these are known as INDEX REGISTERS. Don't worry about these for the moment; all will become clear later in the book.

The A Register

This is often called the ACCUMULATOR because it accumulates the results of many CPU operations. Think of it as the right hand of the CPU; just as many operations are best done with the right hand, the CPU often can only do certain operations with the A Register.

The HL Register Pair

This is a very commonly used Register Pair within the CPU. Look at it as a 16 finger accumulator, or as the "Right foot" of the CPU.

The other registers and register pairs are general purpose ones, with the exception of the F register that we mentioned earlier.

Alternative Registers

Within the CPU there are some more hands that we can use, but only for a very limited range of jobs. These extra hands are called the ALTERNATIVE REGISTER SET of the CPU, and are individually known as A', B', C', D', E', F', H', and L'. There are no alternate Index Registers. The only thing that we can do with them is to copy the contents of the main registers into the Alternate registers for safekeeping while we use the main registers for something else. When we do this, the current contents of the Alternate registers are copied into the main registers. When writing machine code programs for the Amstrad computer, however, these registers are used by the BASIC Interpreter, and so

it's often a good idea not to alter the contents of these registers. (Amstrad recommend that you do not use the alternative register set).

The Stack Pointer

This is a rather specialised CPU foot that points to the address in memory that the stack has grown to. As the stack grows down into memory, the number held on this foot decreases as more information is PUSHED onto the stack. The contents of the Stack Pointer, or SP, are altered whenever the CPU PUSHes or POPs the stack. This register is only rarely manipulated directly in a program.

The Program Counter

The program counter tells the CPU where in memory it can find the next machine language instruction so that the CPU can fetch this instruction and decide what to do. The Program Counter is not directly manipulated in machine language programs.

The fetching of instructions is dealt with by the CONTROL UNIT of the CPU.

The Control Unit

This is the supervisor in the CPU. It coordinates and times the various operations of the CPU, and is responsible for fetching a machine language instruction from memory. The location from which the instruction is fetched by the address held in the Program Counter. The instruction is then passed to a CPU hand called the Instruction Register.

The Instruction Register

This CPU register holds an 8 finger number that represents the machine language instruction that is to be executed next by the CPU. The Control Unit is now responsible for working out what the instruction is and acting upon it.

The Arithmetic and Logic Unit

This is best seen as the pocket calculator of the CPU. It's controlled by the Control Unit of the CPU rather than by a keyboard and is rather simple in what it can do. Addition and Subtraction are easy, but multiplication and division are not possible. It can also compare the values of 8 finger numbers, or perform operations upon the fingers within registers. i.e. it can cause a finger to be raised, or set to '1', or lowered, or set to '0', as required. As a by product of the operations of the ALU, the fingers of the F register are affected.

Although the CPU is a rather clever device, it would be useless without the other devices in the Amstrad. If you examine Figure 2, you'll see that the CPU is connected to virtually all other devices in the computer. Let's now take a look at these other devices and see how they contribute to the operation of the Amstrad Computer.

Memory

Because of its 16-finger Program Counter, the Z80 can gain access to 65536 different locations in memory. However, certain areas of memory within the Amstrad are effectively "used twice" by the computer, and this gives the appearance of the computer being able to access more than this amount of memory. We'll see a little more of this interesting point shortly, but it is rather complex and we'll not go into it in any great detail. There are two different types of memory within the Amstrad; these are called Read Only Memory and Random Access Memory. Don't worry about the jargon; all will be revealed.

Read Only Memory

This type of memory is used in the Amstrad to hold the Operating System and the BASIC Interpreter. This memory keeps its contents even when you turn off the power to the computer. However, the programmer, even by using machine code, cannot alter the contents of this type of memory. We can still load the CPU registers with numbers that are held in Read Only Memory, or ROM, if we wish, or we can run the machine language programs that are stored in ROM. The reason that we call ROM Read Only is therefore obvious; that's all we can do with it!

Random Access Memory

I prefer the unofficial but more descriptive name of Read and Alter Memory for this kind of memory. We can read numbers from it, or we can write new numbers to it. We thus use it to store BASIC programs in, and we'll also use it to write our machine code programs in.

Although we can easily alter the contents of this type of memory, RAM has one very annoying feature. When the power is removed, the memory forgets everything it held before. For this reason, we need to store our programs on cassette tape or disc to keep permanent copies of them. This applies to BASIC or machine code programs.

As we've already mentioned, a location in memory can only hold an 8-finger number. This is true for both RAM and ROM. The range of numbers that can be held in memory locations is therefore 0 to 255. (We'll

mention later how we can use two of these locations to store a 16-finger number).

The Amstrad has 65536 locations of RAM in its memory; it also has over 8000 locations of ROM, the ROM overlapping certain areas of RAM. It's not really important at this point to know what RAM locations are overlapped by ROM, as the Operating System of the Amstrad takes care of what type of memory of CPU "sees" in these overlap locations at particular times. The Practical upshot for machine code programmers on the Amstrad is that we can use the already written machine code programs, present in the ROM, for doing things like printing characters to the screen from machine code. So, let's leave memory alone for the time being and move on to the other parts of the Amstrad System.

The CRTC

The Cathode Ray Tube Controller chip (wow, what a mouthful!) is responsible for getting the information held in a particular part of memory, called the Video RAM, on to the monitor screen as an image.

The PSG

The Programmable Sound Generator is responsible for producing the many sounds that the Amstrad is capable of. We'll look at how we can make sounds from machine code later in this book.

The PPI

The Programmable Peripheral Interface is a vitally important chip in the Amstrad. It has a role in controlling the display, keyboard, PSG, Printer port and cassette recorder. This device acts as a go between the CPU and these other devices.

The Gate Array

This is a clever piece of electronics specially designed for the Amstrad, and it helps the CRTC generate the screen image. It also controls whether the CPU sees ROM or RAM in those locations of memory where the two overlap.

All these devices are controlled by the CPU, which is normally running the Operating System or BASIC Interpreter program in the ROM. We'll later see how we can use routines present in these ROM programs to enable us to control these devices from our own machine code programs. Don't let this frighten you; it's much easier than it sounds, and is certainly an easier job than writing programs from scratch to do these jobs!

We've now examined the "cast list" for the rest of

the book, and we'll later see how we can program these devices to perform various tasks. However, we'll now go on to look at a subject rather fundamental to computing - the subject of counting.

Chapter 2

How Computers Count

I mentioned in Chapter 1 that the CPU can represent numbers between 0 and 255 on one of its 8 fingered hands. How can this be, when we only count to 10 on our fingers? Well, we count on our fingers in a rather inefficient way, and the computer simply uses its fingers more wisely than we do.





When we count on our fingers, we let each finger have the same value. i.e. a raised first finger represents the same value as a raised second finger. There is no reason why this should be so. You could, in fact, use the different fingers to represent different values.

For example, a raised first finger could represent the value '1', a raised second finger '2' and so on. In this scheme of things, therefore, we could represent the number '3' on just two fingers, by raising both the first and second fingers. When either finger is lowered, of course, it has the value '0'. This method is obviously more efficient than our way of counting on our fingers.

Our normal method would, as I mentioned, require 3 fingers to represent the number 3, whereas this new method only needs two fingers. The counting method used by the CPU is based on this idea, and appreciates the below facts;

- (a) That whether a finger is raised or lowered is important to the overall number being represented on the fingers.
- (b) That the position of the finger within in the hand is important to the value represented on that finger, and hence to the value of the number represented on the hand.

Let's take a look at our new method of representing numbers using two fingers.

	Number Represented	
	=	0
	=	1
	=	2
	=	3

We might represent a raised finger by the digit '1' and a lowered finger by the digit '0'. This is easier than drawing pictures of hands all over the place! Thus the above can be rewritten as;

00	=	0
01	=	1
10	=	2
11	=	3

This should look vaguely familiar; remember our way of representing the presence or absence of an electrical signal? We used 1's and 0's there as well.

Such a method of representing numbers in which there are only two different states (raised or lowered finger, 1 or 0) is called a BINARY method of representing numbers. If we add yet another finger to the two we've already considered, then there are 8 different combinations of raised and lowered fingers, including the state when all the fingers are lowered.

If you don't believe this, try it with your own fingers. We can therefore represent the numbers 0 to 7 on these three fingers. Let's use our '0' and '1' representation to represent raised and lowered fingers, remembering that a lowered finger is represented by '0' and a raised finger is represented by 1.

000	=	0
001	=	1
010	=	2
011	=	3
100	=	4
101	=	5
110	=	6
111	=	7

The addition of a fourth finger enables us to represent the numbers between 0 and 15. In computing circles, the numbers 10 to 15 are represented in a

special way by the letters A to F, rather than by the two digit numbers 10 to 15. Thus,

10	=	A
11	=	B
12	=	C
13	=	D
14	=	E
15	=	F

The method of representing numbers in this way is called **HEXADECIMAL** representation. In this representation, therefore, the numbers 0 to 15 are represented as 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. The decimal number 16 is represented as hexadecimal 10, decimal 17 as hexadecimal 11 and so on.

The Amstrad computer allows us to type hexadecimal numbers into the computer in BASIC. However, it's obviously necessary to tell the computer how to distinguish hexadecimal numbers from decimal numbers.

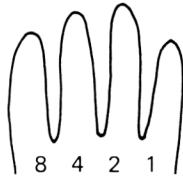
We can give a hexadecimal number either the prefix "&" or "&H". Either of these will be recognised by the Amstrad as the start of a hexadecimal number. There are other methods of designating hexadecimal numbers, and these are shown below. However, the Amstrad only recognises & and &H.

&A	=	10
&HA	=	10
HA	=	10
AH	=	10

Throughout this book, we'll use the & symbol to designate hexadecimal numbers. You can probably see now that using hexadecimal representation to represent machine code instructions is rather convenient. A machine code instruction must be able to be held on 8 fingers, (otherwise it could not be stored in the computer), and we know we can represent 4 fingers as 1 hexadecimal number (0 - F). Thus an 8 fingered number can be represented with only two hexadecimal digits. The advantages of using the hexadecimal representation to represent machine language instructions are therefore;

- (a) We can easily convert hexadecimal numbers into binary numbers, and so we can see which fingers are lowered or raised within a number.
- (b) We can tell, by the numbers of hexadecimal digits in the number whether the number will fit into one or two hands; one handed numbers have 2 digits and two handed numbers have 4 digits.

We do this by remembering that each finger has a different value attached to it.



We've got 4 fingers raised here, and we have assigned each finger a value. When any finger is lowered it has a value of 0. Thus if all the fingers are raised, the fingers will be representing the number

$$8 + 4 + 2 + 1 = 15$$

We simply add up the values that have been assigned to each raised finger. Thus if the left most finger were to be lowered, the value represented would be

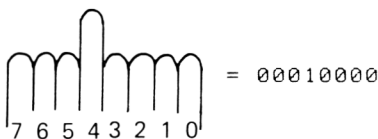
$$0 + 4 + 2 + 1 = 7$$

If you're mathematically inclined, you'll probably note that the value represented by each finger is multiplied by two as we go from right to left. If we number the fingers in the below fashion,



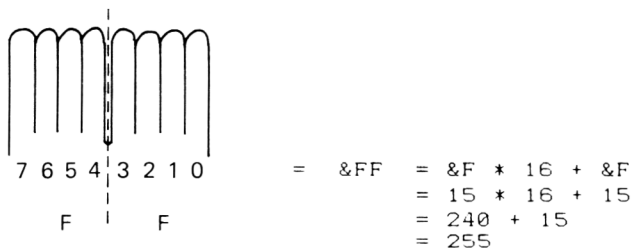
then the values assigned to each finger is 2 to the power of N, where N is the finger number. 2 to the power of 0, for example is 1.

So far we've seen how we can represent numbers that have a value between 0 and 15; you should be able to see what to do to enable us to represent larger numbers; we simply add more fingers. We'll thus end up with the 8 fingered number that a CPU hand is capable of representing; For example, the number 16 is represented on an 8 fingered hand in the below fashion, with finger number 4 raised.



This can be written in hexadecimal as &10. We arrive

at this by splitting the 8 finger number into two 4 finger numbers, and we then give each 4 finger number a separate digit. In this case the right hand 4 fingers are all lowered, thus representing the value 0, and of the left 4 fingers the first finger is raised, thus representing 1. However, the significance of each of these 4 finger "handlets" is not the same to the value of the number overall. The left 4 fingers represent 16 times as much as the right 4 fingers. As a further example, look at the below case, where all 8 fingers are raised.



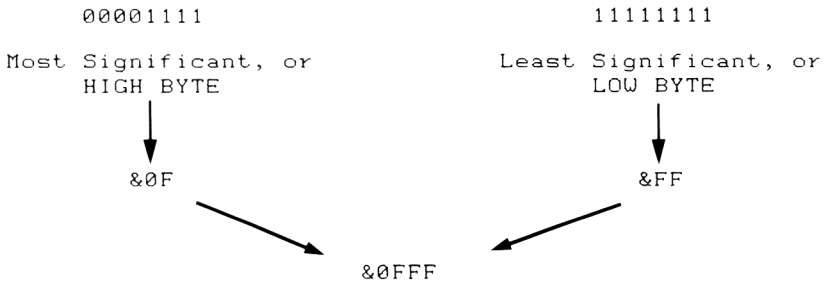
You can thus see how we can represent the number 255 on 8 fingers and how the CPU manages to count to 255 on its 8 fingered hands. An extension of this principle will enable us to count to 65535 on 16 fingers.

Bits and Bytes

It's now time to introduce the proper names of the hands and fingers that are used in computer counting. In common English, an alternative name for a finger is a digit, and it is the same in computing. Each finger, or binary number, is thus called a digit, and there are thus 8 digits in our 8 finger numbers. There is a special name for Binary digits, however, and this is the name BIT. This is a contraction of the phrase Binary Digit. We can thus say that our 8-fingered numbers are 8-bit numbers. These collections of 8 bits are called BYTES. A byte is thus a number that can be represented on 8 fingers or that can be held on a CPU hand. Our 4 finger "handlets" are called nibbles, a nibble, after all, being a small byte...

The terms bit, byte and nibble are very common in computing circles, and so you'll come across them throughout this book and in many other works as well. Just as we numbered our fingers, we number the bits within a byte, and we number them in a similar fashion; that is, 0 to 7 from right to left. Bit 0 is given a special name. It is called the LEAST SIGNIFICANT BIT, or LSB. This is because the value associated with this bit, 1, contributes the least to the value of the number represented on the 8 bits. For a similar reason, bit 7 is called the MOST SIGNIFICANT BIT, or MSB of the byte.

In a similar fashion, we also label the two bytes that make up a 16 finger number of the type that can fit into a CPU "foot" or Register Pair.



Here, the High Byte has a significance to the overall value of the 16 finger number of 256 times the Low Byte. Thus the total value of a number that is held in a 16 finger number is given by

$$\text{value} = 256 * \text{high} + \text{low}$$

where low is the value held in the low byte and high is the value held in the high byte. In a similar fashion, a 16 bit Register Pair in the CPU is said to be made up of a High Register and a Low Register. When you write down a Register Pair's name, the first register that you write down holds the high byte and the second register holds the low byte. Thus in the BC register pair, the B register holds the high byte and the C register holds the low byte. This fact isn't too difficult to remember if you think about the HL register pair; the H register holds the High byte and the L register holds the Low byte. This is probably why the name G wasn't used when the registers were originally named; a GH register would be rather confusing!

Representation of Information

Human beings deal with information mainly in the form of numbers and letters. Computers only deal in numbers. It is thus clear that the computer must have some means of representing other forms of information. There are two main types of information represented by numbers in a computer;

- (a) Machine language programs. These could be the BASIC Interpreter, Operating System or some programs written by the user.
- (b) Data for a machine language program. This can either be numeric, or can include letters. The BASIC program might, for example, be considered as data for the BASIC Interpreter.

Let's now look at how different types of information are represented in the computer memory.

Program Representation

A machine language program is a sequence of bytes that represent instructions to the CPU of the computer. They are stored in the memory of the computer. 260 instructions can be between 1 and 4 bytes in length. For example, the HALT instruction is a single byte instruction. Once the number that represents this instruction has been recovered from the memory and acted upon, then the next instruction is fetched by the CPU. However, if the CPU realises that the number recovered is part of a multi-byte instruction, bytes are fetched successively until the CPU has a full instruction to work on. It is obvious, therefore, that a single byte instruction is executed more quickly than one made up of several bytes.

Data

In BASIC we are able to use various types of variable to hold information on which our programs are to work. These include Integers, Real Numbers and strings of characters. In machine code we don't have this sort of versatility; the only numbers that the CPU can handle directly are integers in the range 0 to 255 or 0 to 65535. To use floating point, or real numbers, such as 1.2345, the CPU has to be programmed appropriately, and the amount of programming required often makes it more advisable to write programs that are to deal with these sort of numbers in BASIC. Character strings are available to us in machine code with only a little extra effort, and the letters that make up strings are represented in the computer by numbers, as we shall soon see.

Integers

We've seen already that integers are easily dealt with by the CPU provided that the numbers are in the range 0 to 65535. However, what about negative numbers? How are SIGNED INTEGERS, as these numbers are known, dealt with in the computer?

Signed Integers

These numbers must obviously be represented in some form of binary representation for the sake of the CPU, so we need a method of representing the SIGN of the number, just as we use the "+" and "-" in normal arithmetic. The most commonly used representation states that a number will be treated as being negative if the "thumb" of the "hand" is raised - i.e. the most significant bit of the number is set to a value of 1. This leaves us with, for an 8 bit number, bits 0 to 6

to represent the actual value of the number. For this reason bit 7 is often called the SIGN BIT of an 8 bit number. You can probably see that the use of only 7 bits to represent the value of the number means that we can no longer represent the number between 0 and 255 any more. Instead, half the numbers represented will be with the sign bit set to 0, and hence will be positive numbers, and half will have the sign bit set to 1, and so will be negative numbers. The new range will be -128 to +127.

This gives us a rather significant problem. How do we tell whether a number is a large positive number or a negative one? The answer is - we don't. It depends upon what interpretation the programmer puts on the numbers at any time. All the machine code instructions will work perfectly well, but the interpretation put on the result depends upon the representation used. Producing a negative number is not, unfortunately, just a matter of setting bit 7 to 1. We must now work out how to represent the actual value of the negative number on bits 0 to 6 of the byte. The fundamental thing to remember about a negative number is that when you add it to the corresponding positive number the result is 0. That is:

$$(-1) + (+1) = 0$$

Thus the representation of -1 must be such that when added to +1 the result is zero.

```
00000001
+1???????
00000000    desired result
```

If we represented -1 by 10000001, then by binary addition we'd set bit 0 of the answer to zero, but what about the other bits of the answer? Would these also be set to zero?

```
00000001
+10000001
10000010    actual result
```

That's not the desired answer. We really need to take the carry that was generated by 1+1 and somehow use it to set all the other bits of the result to zero as well. This requires that all the other bits of the binary representation of -1 should be set to 1 as shown below.

```
00000001
+11111111
00000000    actual result
```

Well, this is certainly the correct answer for -1, but how can we apply similar techniques to other negative numbers? Let's see if we can work out the general rule

for arriving at a representation of negative numbers.

It would appear that the representation of -1 given above was obtained in two stages. The first of these was the replacement of all the 0's in the positive number with 1's and all the 1's of the positive number with 0's, thus representing the number as below.

change 00000001
to 11111110

This process is called COMPLEMENTING the number. The complement of 1 is 0 and that of 0 is 1. The second stage of the process would appear to be adding 1 to the result of the complementing operation. Thus,

$$11111110 + 00000001 = 11111111$$

The only way to see if this method does give us a proper representation of a negative number is to try it and see. Let's see if we can get a binary representation of -2.

(a) Complement +2:

the complement of 00000010
is 11111101

(b) Now add 1 to the complement:

$$11111101 + 00000001 = 11111110$$

This should be the binary representation of -2. To see if we're correct, let's try adding it to +2;

$$\begin{array}{r} 11111110 \\ +00000010 \\ \hline 00000000 \end{array}$$

Yes, that's correct. We get the result zero. This method of representing a negative number in binary is called TWO'S COMPLEMENT representation. This is the most common form of representation for such numbers. If you apply this to a negative number, you'll get the binary representation of the positive counterpart. This isn't really surprising, when we remember that two minuses make a plus!

So far, we've seen this technique applied to 8 bit numbers, but the principles can be applied to 16 bit numbers as well. When we use the Two's Complement notation to represent a negative 16 bit number, the MSB of the High Byte is the sign bit, and bit 7 of the low byte is left alone. The complementing operation, however, is the same for both 16 bit and 8 bit numbers. When we use a 16 bit number in this way, the range of numbers that can be represented is -32768 to

+32767 instead of the usual 0 to 65535.

This is a good place to note that the Integer Variables supported by the Amstrad BASIC are stored as 16 bit Two's Complement numbers, and can thus have values between -32768 and +32767.

Characters and Strings

In machine language programs, we may want the numbers stored in memory or held in CPU registers to represent characters rather than machine language instructions or numeric data for machine language programs. Let's define a character as anything that we can put on the monitor screen with a PRINT command. Thus the collection of letters and numbers in "134 HELLO" are all characters, and they are collectively known as a string of characters, or just a string. As there are no more than 255 characters available to the programmer, the numbers used to represent characters are all 8 bit numbers. How do we decide what numbers are represented by what number? Well, there are a few standards around, and the one used by the Amstrad is called the ASCII code. This stands for "American Standard Code for Information Interchange" and is the most commonly used code for character representation. For example, the code for the letter "A" is the number 65. To see what ASCII code is possessed by a particular character, we can use the BASIC ASC() function. Thus

```
PRINT ASC("A")
```

will print 65 to the monitor.

Summing Up

You can thus see that what a particular number in a memory location or register represents depends mainly upon what the programmer wants it to represent. A number can be any of the following;

- (a) A machine language instruction.
- (b) A number in the range 0 to 255.
- (c) A number in the range -128 to +127.
- (d) Part of a 16 bit number or part of a multi-byte machine code instruction.
- (e) A number representing a character.

It is thus important for the programmer to keep track of what he uses various parts of the computer memory for; it would be disastrous, for example, for the CPU to treat a sequence of bytes representing the message "Hello there!" as a program.

The problem now remains of how to get these numbers into the computer memory, be they machine code programs or data. We use BASIC to help us do this, as well as to help us run our programs. Let's see how.

Chapter 3

Machine Code meets BASIC

As I mentioned earlier on in this book, from the moment we turn on our computer the Z80 CPU is executing the BASIC Interpreter program that is stored in the Amstrad ROM. When we want to tell the CPU to run machine code programs that we've written, we must instruct the CPU to leave the ROM program for a while and go off and run the program that we have written. We have to do this from BASIC, because we are, rather obviously, in the BASIC Interpreter. We must also enter the bytes that make up the machine code program in to the computer memory using BASIC commands. In this Chapter, therefore, I want to examine the commands in BASIC that are invaluable to us when we are writing machine language programs. You may have read of some of them in the Amstrad manual, or they may be totally new to you. Don't worry, we'll examine each command in detail. I'll also look at how we can pass information between BASIC programs and machine language programs. This is very useful, because we often want to write machine language programs that are used in conjunction with BASIC programs to do jobs that cannot be done from BASIC.

Homes for machine code programs

The machine language programs are made up of sequences of bytes representing machine code instructions. The most important thing, therefore, is to find a place in the computer memory where the machine code can live in safety. It's clear that this has to be in RAM memory; don't forget that we can't alter the contents of the ROM. Many areas of the RAM in the Amstrad are not suitable for the storage of the bytes that make up machine code programs. These areas of RAM are those that are used by the computer for storage of the BASIC program, or the BASIC variables, or by the Z80 CPU itself as scrap paper while it executes the BASIC Interpreter or Operating System program. Any machine code programs placed in these areas are thus prone to being overwritten by the BASIC program as you add more lines, by new BASIC variables as the program runs or by the Z80 as it executes it's programs. The areas of the RAM that are used by the CPU while it executes the

programs in the Amstrad ROM are called System Workspace. If we alter any of these memory locations, then we stand a very good chance of altering some of the information that the CPU requires while it is running the Operating System or Interpreter programs. The Z80, not surprisingly, could then lose track of what it's doing and "crash". This is as unpleasant an event as it sounds; we lose all control of the computer and it is often necessary to turn the computer off to regain control of the machine. This rather desperate ploy will result in the loss of whatever is in RAM. The moral of this little story is that when we are experimenting with machine code programs, we should make use of the Amstrad built-in Tape Recorder and save the programs frequently, so that if we have crashed the machine we can simply reload the machine language program from tape and find what caused the problem. This is much easier than typing in the program from scratch!

Crashing the computer will probably be a common occurrence until you find your machine code programming "feet", so don't worry. This is caused mainly by the fact that machine code programming is a rather tedious and error prone task until you get used to it. Machine code programs are also very unforgiving about programming errors. In BASIC, the programmer is given lots of help, with error messages and the opportunity to change the offending program line. There are no such niceties in machine code programming! If you are lucky when you make an error in your machine code program, something unexpected will happen. If You're unlucky, the result could well be a crash!

Anyway, let's return to the problem of finding a home for our machine language programs. It's obvious that this home must be safe from being affected by the action of the BASIC Interpreter or the Operating System of the computer. The safest place for a piece of machine code is made by taking some RAM away from the memory that is available for BASIC programs and variables. We do this by using the MEMORY command in BASIC. Before we see how we use this command, let's look, in a simple way, at how memory in the Amstrad is arranged. Figure 3 shows a simple representation of how the Amstrad memory is arranged. Such a diagram is called a MEMORY MAP. Remember that the address of a byte in memory is simply the location of that byte within the computer memory, and each memory location can hold an 8 bit byte; i.e. a number between 0 and 255.

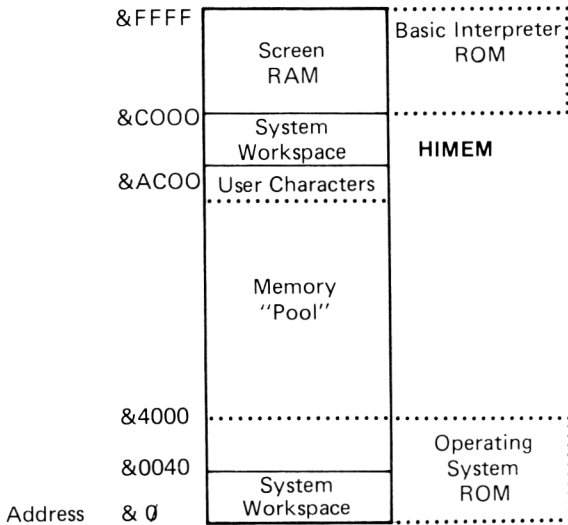


Figure 3. Amstrad Memory Map

Before going any further, let's examine the memory map in a little more detail. The addresses are given in hexadecimal notation. The map shows a simplified version of how the memory is arranged for BASIC programming on the Amstrad. The memory "pool" is that area of RAM that is available to the Interpreter for BASIC programs and variables. The address of the highest numbered memory location that the BASIC Interpreter can use in this way is given a special name; it's called HIMEM. When we turn the machine on, HIMEM is set to a value of &AB7F. The area of memory between HIMEM and address &AC00 is set aside for a special purpose. It is used for storing information about the User Defined Characters that you can create using the SYMBOL command. The position of HIMEM in memory thus changes, depending upon how many User Defined Characters we want to use. It also changes when we use data files on cassette. In fact, when we open a data file from BASIC HIMEM is "moved down" in memory, thus decreasing the amount of space available to the BASIC Interpreter. At any time we can find the value of HIMEM by

```
PRINT HIMEM
```

or

```
PRINT HEX$(HIMEM)
```

The latter will return the value of HIMEM represented as a hexadecimal number. Any memory above HIMEM but

below address &AC00 is thus free from the ravages of the BASIC Interpreter and the Operating System, provided that we don't accidentally clobber the area of memory that is being used to store the information about the User Defined Characters.

So, how do we know where the User Defined Characters start in the area of memory between HIMEM and &AC00? Well, with no user Defined Characters in use, HIMEM is set to &ABFF; thus, as the System Workspace starts at address &AC00, no space is taken up by User Characters when none are in use. However, on turning our Amstrad on, space is set aside for 16 User Defined Characters, and this results in HIMEM being brought down to address &AB7F. Thus, if we were to move HIMEM down even further in memory, the area of RAM between the NEW HIMEM and &AB7F would be free for our machine code programs. &AB80 is the first byte in this instance that is used for storage of User Defined Characters.

The stages of setting aside some memory for our machine code programs are thus as follows.

- (a) Decide how many User Defined Characters you actually want to use in the program. Then use the SYMBOL AFTER command to reserve space for them. Thus, if you wanted to use the User Defined Characters in the range of ASCII codes 200 to 255, you'd execute a SYMBOL AFTER 200 command.
- (b) Now get the value of HIMEM, as shown above.
- (c) Estimate the amount of memory that you want to use for your machine code programs. Call this value N. Now work out

$$\text{NEWHIMEM} = \text{HIMEM} - \text{N}$$

- (d) Finally, use the BASIC instruction MEMORY to tell the computer that we want to alter the value of HIMEM. We want to set it to the value of NEWHIMEM that we've just worked out. The command below will do this, making the byte in memory with the address NEWHIMEM+1 the first byte of memory available to you for machine code programming.
- (e) Thus, to reserve memory starting at, and including, address 40000, you'd issue a

MEMORY 39999

command.

The memory thus reserved is safe from BASIC, provided that we don't overwrite the User Defined Character space. The address of this area of memory can be obtained by making a note of the value of HIMEM present immediately after we'd used SYMBOL AFTER. Remember that this will reduce the amount of memory

can be obtained by making a note of the value of HIMEM present immediately after we'd used SYMBOL AFTER. Remember that this will reduce the amount of memory available to BASIC, but, as they say, you can't make an omelette without breaking eggs!

While we are talking about memory use, a slight detour is in order to look at a little jargon that you will come across with regards to computer memory. It's basically a way of getting out of writing long strings of numbers when we are talking about how much memory in a computer is available for particular purposes. In the Amstrad, there are 65536 locations of RAM. You will often come across the abbreviation 64K for this quantity of memory. Thus when we say that there is 64K of memory in the machine we actually mean that there are 65536 different locations of memory in the machine. Similarly, 2K is 2048 bytes and 1/2K is 512 bytes.

After that short detour, let's carry on with our examination of useful BASIC commands for machine code programs. We can now find a place in RAM for the bytes that make up our machine code programs to live in, but how can we actually get the bytes into the memory? Also, can we directly examine the contents of a memory location? The answer to both questions is "Yes".

POKE and PEEK

When we want to put a byte into a particular memory location, we use the rather descriptively named command POKE. This is exactly what the command does; it puts, or POKES, a particular byte into a particular memory location. We use it in the following fashion;

```
POKE address,value
```

where address is the address of the location that we want to POKE the byte into, and value is the value of the byte we want to put in that location. Thus the command

```
POKE 40000,210
```

will put the number 201 into location 40000 of the Amstrad RAM.

Although it's possible to POKE any address in the computer, TAKE CARE! POKEing System Workspace, which is very ill advised, can easily cause a crash. Similarly, POKEing parts of the RAM that are storing BASIC program lines or BASIC variables could lead to either the BASIC Interpreter losing track of parts of the program, or variables being altered in value. You have been warned!

Having used the POKE command to put the bytes that represent our machine code program into memory, it

would be rather nice to be able to look in to the memory and find out what value is held in a particular memory location. This is done by using the PEEK command, which enables us to PEEK into a particular memory location to see what value is in there.

```
PRINT PEEK(address)
```

will print to the screen the value of the byte held in memory location with the address 'address'. Thus

```
PRINT PEEK(40000)
```

will return the value currently held in location 40000. An alternate use of this command allows us to set a BASIC variable to hold the value PEEKed from a memory location, as in

```
LET A=PEEK(40000)
```

In all PEEK and POKE instructions, the parameters, which are the numbers we follow the command with, must be whole numbers. In the POKE instruction, the 'value' parameter should be in the range 0 to 255. In both cases, the address should be in the range 0 to 65535. Also, you can use BASIC variables in POKE and PEEK statements. Thus the commands

```
LET address=40000:LET value=64:POKE address,value
```

would result in the contents of address 40000 being set to 64. You can see that the BASIC commands PEEK and POKE are at the very heart of writing machine code programs on our Amstrad computer. Without them, we wouldn't be able to enter the bytes that make up the programs into memory, and so we wouldn't be able to run the machine code programs.

Speaking of running machine language programs, how can we actually get the CPU to execute the machine code instructions once we have POKED them into place? The CPU will still be executing the BASIC Interpreter program, so we really need a BASIC command that causes the CPU to leave the Interpreter, execute a machine code program that is resident at a certain address within the computer memory and then return to BASIC. Well, not surprisingly, there is a BASIC instruction to do this. It is called CALL.

CALL

The BASIC Interpreter treats your machine code programs as machine code subroutines. Just as in BASIC, where you can have subroutines to perform particular jobs, you can have them in machine code programming, as we shall learn later in the book. Suffice it say for the moment, that the CALL command effectively tells the BASIC Interpreter to do a GOSUB

to the machine code program that you want running. Obviously, the Z80 doesn't actually execute a BASIC GOSUB instruction, but does the machine code equivalent of it. The CPU must obviously be told of the address in memory of the first machine code instruction that it is to execute. This is done by passing a PARAMETER to the CALL command. This parameter is a whole number in the range 0 to 65535, and it specifies the address of the first instruction in your machine code program. For example, try typing in the command below, then press ENTER.

```
CALL &BB18
```

This causes the CPU to execute the machine language program whose first instruction is at address &BB18. At that address, there is a ROM machine language program which causes the machine to wait until a key is pressed before going on. You might be able to think of a use for this command in your BASIC programs.

This way of using CALL is the simplest way of running a machine code program from BASIC; simply give the CALL command the address of the machine code program of interest and away we go. However, there are a couple of disadvantages with this method.

- (a) There's no obvious way of passing information from BASIC to the machine code program.
- (b) There appears to be no method of passing information back to BASIC from the machine code programs.

These two disadvantages are not terribly important when we start learning machine code, but they become very important as soon as we want to write machine code programs that do jobs that we cannot do from BASIC; in these situations, it's often useful to be able to tell the machine code program the value of BASIC variables, and also useful to get information back from the machine code programs into BASIC variables. To do this, we simply use an extended version of the CALL command. An example of this is

```
CALL 40000,3,4,5
```

This would call the machine language program at address 40000 and would make the numbers 3,4 and 5 accessible to that program. A similar command is

```
CALL 40000,A%,G%,F%
```

This would make the values held in the 3 BASIC variables A%, G% and F% accessible to the programmer. The values that we make accessible to the machine code programs in this way are called parameters, and we say that we're PASSING PARAMETERS to the machine language program when we do this. We will have a detailed look

at how the Amstrad BASIC Interpreter allows us to pass parameters to and from our programs in Chapter 5. If we were to look at this subject now, we'd be jumping ahead of ourselves a little bit. Let's get back to reviewing our BASIC commands.

Saving bytes on tape

When we save our BASIC programs to tape, we use the SAVE command in BASIC. We use a similar command to save the bytes that make up our machine language instructions to tape. The command we use is

```
SAVE filename,B,start,length,execution_address
```

Let's examine each part of this command. The filename is a string which will give the file on tape its name. It will be a string constant, such as "PROG". The B is essential; this causes the computer to save a specific area of memory to tape, rather than just the BASIC program. The area of memory that we want saving to tape is specified by the start and length parameters as follows.

start This is the address of the first byte of memory that we want saving to tape.

length This is rather self-explanatory, and is the number of bytes that we want to save on tape.

Both of these can be constants or can include variables. They should both be in the range 0 to 65535. The final parameter is optional - that is, we don't have to use it unless we want to. The execution address of the file is the address in the block of memory saved that will be treated as the address of the first instruction of the program if we RUN the file. If this parameter is left out when we save the file, and you later try and RUN the file, you'll simply reset the whole computer, just as if you'd pressed SHIFT-CTRL-ESC!

RUNning a file

This is the same as the RUN filename command in BASIC. The computer searches for the file with the given filename on the tape, and if it finds it loads it and runs it from the execution address that was saved with the file. Thus

```
RUN "fred"
```

will look for a file called "fred" on the tape and attempt to load it in and run the machine language program that it represents.

Loading a file

It's often more useful to be able to LOAD a file in to the computer rather than load it in AND run it. We can then modify the program if we desire, and then run it with a CALL command. Or, it could be a piece of machine code that is called from within a BASIC program to do a particular job with certain BASIC variables that would be passed as parameters from a CALL command. The version of the LOAD command that we're interested in is as follows;

```
LOAD filename,address
```

The filename is the name of the file that we're looking for, and the address parameter is optional. If we omit the address parameter, then the file is loaded to the address from which it was written. If the address parameter is provided, then the file will be loaded to this address. Thus the command

```
LOAD "FRED",40000
```

will load the file "FRED" to address 40000. There is one point to note about this LOAD command, and that is that when using it files saved with the B parameter can only be reloaded into memory OUTSIDE that area of memory available to BASIC. This means, to us, that the file must be reloaded above the current HIMEM. Problems can arise here, therefore, if you SAVE a file with HIMEM set at one value, and try to reload the file to the same address BUT with HIMEM set higher. In this case, you must provide the LOAD command with an address parameter that is above the current setting of HIMEM, or change HIMEM. (Note that you cannot always move machine language programs to different areas of memory).

There are various ways available to us to enter the bytes that make up our machine code programs into the proper area of memory. The way that I have used to test the programs listed in this book is to put the numbers that represent the machine language instructions into a DATA statement, and then use READ to get the values. They are then POKED into memory. A simple program that can do this is listed below.

```
10 RESTORE 50
20 FOR I=0 TO number:REM number is number of bytes
30 READ A$: POKE(address+I),VAL("&" +A$)
40 NEXT
50 DATA hexadecimal bytes representing
    the machine code instructions.
```

Before using this program, you'll need to add a line that sets up the variables 'address' and 'number'. 'address' is the address of the first byte that you want to POKE into memory. 'number' is the

number of bytes that you want to put into memory. The hexadecimal numbers in the DATA statement must NOT have the "&" prefix; this is added in line 30 of the program. Adding the prefix in this way means less typing for you when you are typing that data in. Thus the number 255 would be represented in the DATA statement as "FF". Each number in the DATA statement is separated from the others by a ",", as is usual in DATA statements. One advantage of placing the bytes in a DATA statement is that they can be saved as part of a BASIC program, and the DATA statement can be edited in the same way as any other BASIC line.

Well, you'll be glad to know that we're just about ready to examine some 280 Machine Language instructions. Before we do this, however, I'll introduce you to what is quite possibly the most important machine code instruction that you'll encounter. It enables us to return to BASIC when we've finished running our machine code program. It has the mnemonic

RET

which is short for RETURN from subroutine, and this instruction is the equivalent of the BASIC RETURN. We said earlier on that the CALL instructions was a bit like a GOSUB; here's the corresponding RETURN.

However, forgetting a RET in machine code can be absolutely FATAL! A crash will be the usual result.

Chapter 4

Registers at work

Well, we're now ready to look at some machine code instructions. It's probably clear to you by now that we cannot do much programming of the Z80 CPU until we can actually get numbers into the CPU registers, and into the memory of the machine from the CPU. So, in this Chapter we'll examine the instructions that we can use to transfer 8 bit numbers between the memory of the computer and the CPU registers, as well as the commands that we use to transfer 8 bit numbers between the different registers of the CPU. We'll take a look at the instructions that we use to transfer 16 bit numbers around the machine in a later Chapter.

As always in computers, there is a little jargon to come to terms with. When we are transferring data from one register to another register or memory location, the register from which the data is being copied is called the SOURCE REGISTER. That place to which the data is being copied is called the DESTINATION REGISTER. Similar terms are used when we are moving data between registers and memory locations. We say that we are loading a register or memory location from somewhere else. Indeed, the mnemonic for these transfers in assembler language, LD, is simply an abbreviation of the word Load.

The designers of the CPU gave us many different ways of handling transfers of data from place to place in the computer, and also several different ways of actually carrying out various operations, such as addition, on data. The ways in which the CPU uses its registers are called ADDRESSING MODES and virtually all instructions in the Z80 Instruction Set use one or other of the Addressing Modes that we'll discuss in this Chapter.

The Addressing Modes will be examined with regard to the LD instructions, but they are applicable to other instructions as well, as we shall later see. From now on we'll write our machine code programs down using mnemonics, for clarity. There will be example routines for you to enter into your Amstrad Computer in later Chapters, and so the mnemonics that make up these programs will have to be translated into machine

code by using the Tables in the back of the book. So, let's get working on the Addressing Modes.

Register Addressing

This is probably the simplest addressing mode available to us for transferring data between registers. It is just the transfer of data from one register to another register. An example of this Addressing Mode is

```
LD   A,B           read this as load A with B
```

This will copy the number that is currently in the B register into the A register. This makes the B register the Source register and the A register the Destination register.

A couple of points to note about this transfer: the first is that the contents of the B register are totally unaffected by the transfer operation. The second is that the contents of the A register before the transfer are, not surprisingly, totally lost. There is a general way of writing down a register to register transfer command. This is

```
LD   r1,r2         load r1 with r2
```

where r1 and r2 are any 8 bit register except the F register. Each of the various possible transfers is represented by a single byte in our machine code programs. For example, the transfer

```
LD   A,C           load A with C
```

is represented by the number &79. There are many such transfer commands, as you can see if you look at the Tables in the back of the book.

These transfer commands are a bit like the BASIC commands of the form

```
LET A=B
```

These are very useful, but we haven't yet seen how we can actually load a register with a particular number. The BASIC command to do this is of the form

```
LET A=7
```

This sets the variable A to hold the value 7. Let's look at the addressing mode that enables us to do this.

Immediate Addressing

This is another addressing mode. This enables us to load an 8 bit register with a number between 0 and 255. The number that is to be placed in a register is

specified as a part of the command. So, this command takes up two bytes in our machine code programs; the first byte is used to specify the actual register that we want to use and the second byte specifies the number we want to put in that register. The general form of these commands is

```
LD   r,n           load r with n
```

where *r* is an 8 bit register, with the exception of the *F* register, and *n* is the number that we want to put in that register. As a more specific example,

```
LD   A,23
```

will put the number 23 into the *A* register. This is represented in machine code by the two bytes

```
3E 17
```

Note that I've put '23' into hexadecimal representation. The first byte, 3E, represents the actual LD *A,n* instruction and the second byte represents the number that we want to put into the register. These two bytes are given special names. The first byte that specifies the actual operation to be carried out by the CPU is called the INSTRUCTION CODE or OP CODE. The latter is short for OPERATION code. The second byte above is called the OPERAND BYTE. Using Immediate Addressing, therefore, we can actually put particular numbers in the CPU registers.

So far, we've only dealt with data transfers involving the CPU registers. We haven't yet moved the data between the CPU registers and the memory of the computer. We'll now look at the addressing modes that enable us to transfer data between the registers and the memory of the computer.

Register Indirect Addressing

Things now begin to look a little more complicated, but the instructions that use this addressing mode are very powerful. Data is transferred between the CPU registers and memory using the HL, BC or DE register pairs to tell the CPU where the data is to be transferred to. Let's take a closer look at this. We have to set up the register pair involved with the address of the location in memory that we want to look at, and we'll see how we can write data to these 16 bit register pairs later in the book. The general forms of these instructions are as follows

```
LD   A,(rr)
LD   (rr),A
LD   (HL),n
```

Here, *rr* is one of the 16 bit register pairs that are

listed above, and n is an 8 bit number. There are a couple of points to look at with regard to these instructions.

- (a) The brackets surrounding the register pair indicates to us that we are interested in the CONTENTS of the register pair; this is true in all Z80 assembler instructions. Whenever we see the brackets, remember that we are interested in the contents of the register pair or memory location in the brackets.
- (b) The HL register pair is already showing itself to be more versatile than the other 16 bit register pairs in that we can load a memory location whose address is in the HL register pairs directly with an 8 bit number. With the other register pairs being used in this way it is necessary to put the number in to the A register first.

This method of using the HL register to directly load a memory location with a number is given a special name. It's an addressing mode called, wait for it... Register Indirect Immediate Addressing. You can probably see that the Register Indirect part of the name comes from the fact that we use a 16 bit register to hold the address of the memory location of interest, and the Immediate part of the name comes from the fact that we're using a number in the command, just as in Immediate addressing. The Register Indirect Addressing commands are all represented by 1 byte in our machine code programs. The

```
LD    (HL),n
```

command requires two bytes in machine code programs; a one byte opcode and a one byte operand.

Using the instructions that we've looked at so far, let's see if we can write a small piece of machine code to transfer the contents of memory location 40000 to location 40001. Don't forget that all the numbers below are in hexadecimal. Thus, 40000 is hexadecimal is &9C40 and so 40001 is &9C41. Lets first look at the series of machine code instructions that we'll use to do this job. They're listed below along with the bytes that we'd have to enter in to the computer.

LD	H,&9C	26 9C	load HL with 40000
LD	L,&40	2E 40	note that 40000 is &9C40
LD	A,(HL)	7E	load A with the contents of location 40000
LD	H,&9C	26 9C	load HL with 40001
LD	L,&41	2E 41	40001 = &9C41
LD	(HL),A	77	load location 40000 with the contents of A
RET		C9	return to BASIC

Let's look at the role played by each of the instructions. The first two instructions put the address of location 40000 into the HL register pair. The third instruction loads the A register from the location addressed by the HL register pair. We then set the HL register pair to hold the address of location 40001, and we then transfer the contents of the A register to the location addressed by the HL register pair, now location 40001. The final instruction, RET, brings us back to BASIC.

The BASIC program below will actually enter into the computer the bytes that make up this machine language program.

```

10 MEMORY 39999
20 FOR I=0 TO 10
30 READ A$: POKE (40002+I),VAL("&" +A$)
40 NEXT I
50 DATA 26,9C,2E,40,7E,26,9C,2E,41,77,C9

```

Line 10 reserves the memory from location 40000 onwards. The bytes that make up the programs are then POKEd into memory from location 40002 onwards. Remember that we're using locations 40000 and 40001 for data for our machine code program. To use the program that the above program enters into memory, POKE a suitable value into address 40000, then CALL 40002, then use PRINT PEEK (40001) to see if the value has been transferred from location 40000 to location 40001. This might seem a trivial example, but shows the basic principles of entering a small machine code program into the Amstrad memory. It's clear from the above that it would have made things a little easier if we could have loaded the A register DIRECTLY from the memory locations, without putting the address of the location into the HL register pair first. Well, there is an addressing mode that allows us to do this.

Extended Addressing

These commands are of the form

LD	A,(nn)	load A with the contents of nn
LD	(nn),A	load nn with the contents of A

where nn is a 16 bit number that represents the address of a memory location in the computer. Typical

instructions using this addressing mode are as follows;

```
LD  A,(40000)
LD  (40001),A
```

These two commands could obviously be used in the example program that we've seen above. As you remember, the 16 bit numbers are held in two bytes; as the above command has a single byte for its opcode, the command requires three bytes to represent it. The opcode goes into memory first, but how do we put the two bytes representing the address into memory? The most obvious way of doing this would be to write the two bytes that represent the address with the High Byte first. Thus,

```
LD  A,(40000)
```

might be represented as

```
3A 9C 40
```

however, this is NOT the way in which the Z80 CPU expects to find the addresses to be used in these instructions. The CPU expects numbers like this to be put into memory with the Low Byte first. The correct representation of the above command would thus be;

```
3A 40 9C
```

This is a rather important point in machine language programming to remember; in a situation where a 16 bit number is to be stored in two memory locations in the computer, the low byte of the number is stored in the lowest numbered memory location and the high byte of the number is stored in the higher numbered memory location. This may seem a little peculiar, but the CPU manages to work things out. All we have to remember is to put the two bytes that make up the number into the memory locations in the correct order.

The Low byte of any number, and the high byte, can be worked out using the below small program.

```
10 INPUT number
20 A$=HEX$(number)
30 sp$=STRING$(4-LEN(A$),"0")
40 A$=sp$+A$
50 PRINT "High byte is "; LEFT$(A$,2)
60 PRINT "Low byte is "; RIGHT$(A$,2)
```

Simply run the program and enter the decimal number that you wish to convert to an upper and lower byte. One point to remember is that in instructions like this, where the CPU is expecting an address, two bytes MUST be entered, even if the address that we want to use in the command is less than 255 in value. Thus the

High byte of 255 is 0 and the Low byte is &FF. The instruction LD A,&FF) would thus be written as the below series of bytes;

```
3A FF 00
```

The CPU is expecting a two byte address in this instruction; if you forget it, a crash is often the result, or a peculiar result from your program. The reason for this is that the CPU will take a byte from the next instruction to "make the numbers up", thus leading to the rest of the program being totally misunderstood by the CPU!

Incidentally, it's easy to work out exactly what the value of the address is when it's represented in this way in the computer memory. Simply work out

```
value = (value held in high numbered byte) * 256
        + (value held in low numbered byte)
```

Thus, if location 40000 and 40001 are known to be holding a 16 bit number, and location 40000 is holding the value &A0 and location 40001 is holding &6F,

```
value = &6F * 256 + &A0
       = 111 * 256 + 160
       = 28416 + 160
       = 28576
```

Let's now return to the addressing modes. The opcodes and the operands for the commands discussed above are as follows;

```
LD  A,(nn)      3A LB HB
LD  (nn),A      32 LB HB
```

where LB and HB stand for the low byte and high byte of the address nn respectively.

To see a concrete example of this instruction in use, the three instructions

```
LD  H,&9C
LD  L,&40
LD  A,(HL)
```

could be replaced by

```
LD  A,(&9C40)
```

which would be represented by the bytes

```
3A 40 9C
```

On executing this command, the A register would be given a copy of whatever data is in location &9C40 of

the computer memory. Similarly,

```
LD (&9C41),A
```

would copy the contents of the A register into location &9C41.

Labels in Machine Code

When we are writing programs in Assembler Language, it often gets tedious to have to remember the exact locations in memory that we've used for storage. To make life easier for us, we often give commonly used locations special names of our own choosing. These names are called LABELS. Thus we might call location 40000, (&9C40) "FRED". We could thus write

```
LD A,(FRED)
```

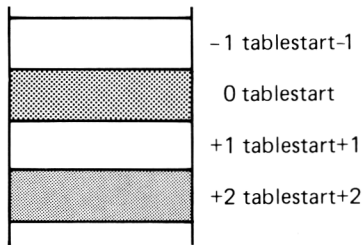
As long as we remember to put the actual address in when we hand assemble the code, we're alright. The use of these names, especially if the names have some relevance to the data that we're putting in these locations, can make an assembler listing more readable. If you eventually get an Assembler program for the computer, you'll be able to use labels with this program.

Indexed Addressing

Remember the IX and IY "feet"? Well, this addressing mode makes use of these registers. As with all the addressing modes that we've seen so far, the Indexed Addressing mode is used by several different types of instruction, not just the load instructions. The Indexed Addressing instructions that we use to transfer 8 bit numbers are of the below form;

```
LD r,(IX+d)
LD (IX+d),r
LD r,(IY+d)
LD (IY+d),r
```

where r is, as usual, one of the 8 bit registers (with the exception of the F register). d is an 8 bit number represented in Two's Complement representation. It is called the DISPLACEMENT byte, and is thus a number between -128 and +127. To see the significance of the displacement byte, look at the below diagram that represents an area of memory. Assume that the IX register has already been loaded with the address of the memory location that's labeled "tablestart" on the diagram.



LD A,(IX+2) copies the contents of location "tablestart+2" into register A

The displacement byte is thus a value that is added to the address that is held in the Index Register to give the actual address of interest. These Indexed addressing commands are thus very useful for accessing data that is stored as "tables" of bytes in memory. These instructions have a 2 byte opcode, and a further byte that represents the displacement byte. Thus the instruction

LD A,(IX+2)

is coded as

DD 7E 02

02 is the displacement byte, and the bytes DD and 7E are the opcode for this instruction. To load the A register from memory location "tablestart-1", we'd have to make the displacement byte equal to -1. All that we need to do is to put the displacement byte in Two's Complement representation. As the Two's Complement representation of -1 is &FF, the command

LD A,(IX-1)

is coded as

DD 7E FF

The opcodes for the various Indexed Addressing instructions will be found in the back of the book.

You may remember that it was possible to load a particular number into a memory location whose address was held in the HL register pair. Well, something similar is possible with the Index Registers. It is a new Addressing Mode, called Immediate Indexed Addressing.

Immediate Indexed Addressing

This simply loads a memory location whose address is specified by the contents of an Index Register and the displacement byte. The general form of these instructions is:

```
LD (IY+d),n
LD (IX+d),n
```

where n is an 8 bit number and d is the displacement byte. An example is the instruction

```
LD (IX+2),&FE
```

which is coded as

```
DD 36 02 FE
```

This instruction has a two byte opcode, and then two extra bytes for the displacement byte and the number that we want to load in to that memory location specified by IX and the displacement.

That just about completes this survey of 8 bit data transfers and the Addressing Modes associated with them. In later Chapters we'll look at 16 bit data transfers, to complete the picture. However, we can now go on to look at the complicated version of the CALL command from BASIC. Remember how it enabled us to make BASIC variables accessible to our machine code programs? Well, the reason I didn't cover this command in the last Chapter was that to understand how it works, we needed to know something about the Index Registers and how they work. We now know enough about this, so in Chapter 5 we'll look at this extended version of the CALL command to see how we can pass parameters to our machine code programs.

Mnemonic	Bytes	Time Taken	Effect on flags						
			C	Z	P/V	S	N	H	
LD register,register	1	4	-	-	-	-	-	-	
LD register,number	2	7	-	-	-	-	-	-	
LD A,(address)	3	13	-	-	-	-	-	-	
LD (address),A	3	13	-	-	-	-	-	-	
LD register,(HL)	1	7	-	-	-	-	-	-	
LD A,(BC)	1	7	-	-	-	-	-	-	
LD A,(DE)	1	7	-	-	-	-	-	-	
LD (HL),register	1	7	-	-	-	-	-	-	
LD (BC),A	1	7	-	-	-	-	-	-	
LD (DE),A	1	7	-	-	-	-	-	-	
LD register,(IX+d)	3	19	-	-	-	-	-	-	
LD register,(IY+d)	3	19	-	-	-	-	-	-	
LD (IX+d),register	3	19	-	-	-	-	-	-	
LD (IY+d),register	3	19	-	-	-	-	-	-	
LD (HL),number	2	10	-	-	-	-	-	-	
LD (IX+d),number	4	19	-	-	-	-	-	-	
LD (IY+d),number	4	19	-	-	-	-	-	-	
Flags notation:									
# indicates flag is altered by operation									
0 indicates flag is set to 0									
1 indicates flag is set to 1									
- indicates flag is unaffected									

Table 1. 8 bit data transfer instructions

Chapter 5

Passing Parameters to programs

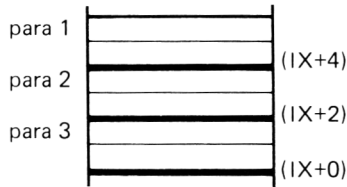
In Chapter 3, I suggested that you could use a special version of the CALL command to pass the values held in BASIC variables over to your machine language programs. In this Chapter we'll examine this use of the CALL command to see how we can pass numbers between BASIC and Machine language programs.

Although it's possible to pass REAL numbers, such as 1.234 or 1E7 over to machine code programs, I will not cover this subject here. Instead, I will concentrate on passing Integers and Strings over to machine code programs, and on passing Integer values back to BASIC from machine code programs.

There are three broad classes of parameters that can be passed over in the CALL statement. You can, of course, mix up these classes of parameter in a single CALL statement. The three classes are;

- (a) A number, such as 100 or 2, or an Integer variable name or an integer expression. The value passed over to the machine code program must be in the range of numbers that can be represented as a Two's Complement, 16 bit number. An example of this type of CALL is CALL 40000,A% where 40000 is the address of the routine and A% is the parameter.
- (b) An Integer variable name prefixed with the "@" symbol, such as @A%. This, as you will soon see, enables us to pass Integer values BACK to BASIC variables from machine code programs.
- (c) A string variable name prefixed by "@", such as @A\$. This is how string variables are passed over.

Now, how do we get access to these parameters that are passed over in the CALL statement from within our machine code program? Well, the IX register "foot" is set up by the BASIC Interpreter to hold the ADDRESS of the low byte of the LAST parameter that was passed over in the CALL command. The below diagram shows this for the CALL 40000,para1,para2,para3 command. The parameters are all passed over as two byte values.



It might seem a little peculiar for the IX register pair to point to the last parameter that was passed over, but you'll soon get used to it. It's clear, therefore, that we can use Indexed Addressing mode instructions to get values from the above memory locations into CPU registers. The area of memory that is pointed to by the IX register is called a PARAMETER BLOCK; the name is fairly descriptive, as it's an area, or block, of memory that holds parameters!

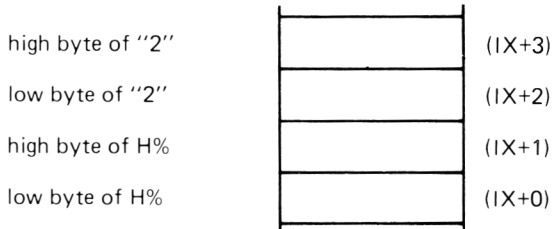
The actual contents of each entry in the parameter block corresponding to the parameter passed over in the CALL statement depends upon the class of parameter involved. We saw above that there were three different classes of parameter in which we're interested; let's now take a closer look at each of these classes.

Integer variables and Numbers

An example of this type of parameter is A% or 5.

On entering your machine code program, then A register will contain the number of parameters that have been included in the CALL command which caused the program to be entered. Therefore, if you want this information, don't forget to store it away before you start using the A register!

For this class of parameter, each two byte entry in the parameter block will contain a 16 bit, Two's Complement number that corresponds to the parameter that was passed over. Thus if the parameter in the CALL statement was the number "54", the appropriate entry in the parameter block would be a two byte number with the value 54. The numbers are stored in the parameter block with their low bytes first. Thus, for the command CALL 40000,2,H% would generate the below parameter block;



Thus to load the value of the variable H% into the HL register pair we might use the below instructions

Example 1:

```
LD  L,(IX+0)
LD  H,(IX+1)
```

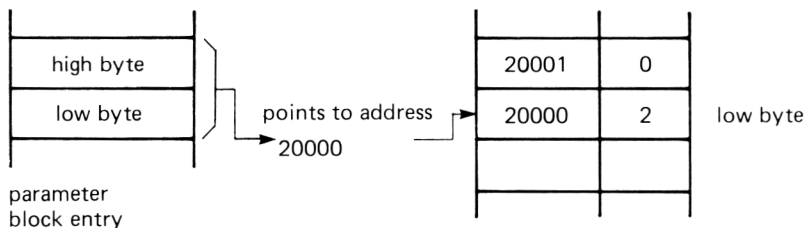
Easy, isn't it? A useful trick to remember is that if you only want to pass numbers to your machine code in the range 0 to 255, you only need to load a register from the low byte of the appropriate parameter block entry.

Variables prefixed with @

If a variable name is prefixed by the @ symbol, such as

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

then the two byte entry in the parameter block that corresponds to this parameter is the ADDRESS of the variable in the computer memory. Thus, if the parameter block entry for @H% had the value 20000 it indicates that the variable H% is stored at address 20000 and NOT that the variable holds the value 20000. So, if H%=2, and we issued a CALL statement containing @H%, the resulting parameter block would point to the address of the H% variable in the following fashion.



One nice thing to note about the use of "@" is that it enables us to actually alter the value held in a variable from within a machine code program and then pass the altered value of the variable back to BASIC where it can be used by the BASIC program. How do we do this? Well, we must write our machine code program so that it puts the value that we want returning in the BASIC variable into the memory locations given in the parameter block entry. Thus, if we wanted to set H% above to the value 7, we'd load location 20000 with 7 and 20001 with 0 before returning to BASIC with the RET statement.

The BASIC Interpreter is usually quite forgiving if you use a variable without first setting it to a particular value. It usually assumes the variable to be set to zero, as you probably know. However, in this use of variables, the variable must have been set up in some way. This is because the BASIC Interpreter

needs to pass into the parameter block the address of the variable concerned; if the variable hasn't been previously used, the Interpreter obviously won't be able to find an address for it! Thus any variable that is used with the "@" character should be initialised in some way before use. If we are passing a parameter into the program in this variable then all will be well, as you'd set the variable to a value anyway. If, however, you're using the "@" variable to return a value from machine code to BASIC, and you're NOT passing a value over to the machine code program, then it's a good idea to set the variable to 0 first. If you fail to do this, you'll get an error message. Let's see an example of the use of "@" in a program. Once the below program is entered,

```
CALL 40000,@A%,value
```

will set the variable A% to the number passed over as the second parameter. The second parameter should be less than 256 in value. A% must be set to 0 before use.

Example 2:

```
LD  A,(IX+0)    DD 7E 00  get second parameter
LD  L,(IX+2)    DD 6E 02  get address of the
LD  H,(IX+3)    DD 66 03  A% variable
LD  (HL),A      77       set A% to value
RET                    C9       return to BASIC
```

The below BASIC program will POKE the bytes in to memory at address 40000.

```
10 MEMORY 39999
20 FOR I=0 TO 10
30 READ A$:POKE (40000+1),VAL("&"+A$)
40 NEXT I
50 DATA DD,7E,00,DD,6E,02,DD,66,03,77,C9
```

To see it in action, run the above program and then type in

```
A%=0:CALL 40000,@A%,4:PRINT A%
```

You should get the value 4 printed to the screen. Try the routine out with other integer variables and other values. Again, remember that the variable must have been previously used so that the BASIC Interpreter knows where to find the address of the variable.

Do you see how the program works? The first instruction gets the low byte of the second parameter into the A register. We then get the next parameter, which is the address of the integer variable used, into the HL register pair. Finally, we use a Register Indirect Addressing mode instruction to put the correct value into the variable. (LD (HL),A is used

here). Finally we make the return to BASIC with a RET instruction.

Passing Strings

Whereas we can pass numeric constants, such as "1" or "3000" over to machine code programs, you can't pass over string constants, such as "UUU". Thus,

```
CALL 40000,@A$
```

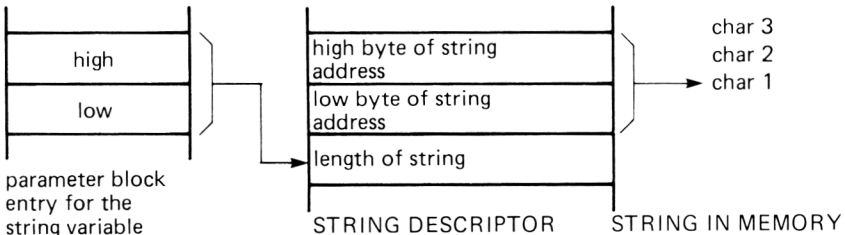
is legal, but

```
CALL 40000,@"fred"
```

is not. Also, the command CALL 40000,A\$ will generate a "Type Mismatch" error. However, we've already said that the parameters are stored in the parameter block as a series of two byte integers. How can we store strings in the parameter block, when a string can be up to 255 characters, and hence 255 bytes, in length? Well, the answer is indicated to us by the fact that strings can only be passed over to a machine code program by prefixing their variable names with the "@" symbol. When we used this symbol above, with a numeric variable, it placed in the parameter block an entry representing the address in the computer memory of the variable. A similar situation arises in this case. However, the address placed in the parameter block is NOT the address of the string itself. It is the address of another block of memory that gives details about the string. In fact, it describes the string, and for this reason is often called the STRING DESCRIPTOR BLOCK, or just the STRING DESCRIPTOR. Let's take a closer look at this area of memory.

String Descriptor Block

The below diagram illustrates the role of the string Descriptor Block.



The role of the Descriptor is thus to

- (a) tell us the length of the string. If the length entry in the Descriptor block is equal to 0 then the string is "empty".

(b) tell us where in the memory the bytes that represent the characters in the string can be found. The address held in the last two bytes of the Descriptor Block is stored low byte first, and is the address of the first character of the string.

As an example of how we can access the string descriptor block, let's write a machine code program that performs the role of the BASIC LEN() function. One thing to note is that any string passed to a machine code program using the "@" symbol must have been initialised in some way, or otherwise the BASIC Interpreter won't be able to give the parameter block the address of the Descriptor Block. So, on with the example.

Enter the bytes that make up the machine code program at address 40000. I'll leave it to you to sort the bytes out, using the Tables in the back of the book. You can use a BASIC program similar to that used in Example 2 above, but remember to change the number in the FOR ... statement to the actual number of bytes in the program. To help you here, there are 15 bytes in the program, and the FOR statement should read FOR I=0 TO 14.

Example 3:

```
LD L,(IX+0)    get the address of
LD H,(IX+1)    the string descriptor
LD A,(HL)      get string length
LD L,(IX+2)    get the address of
LD H,(IX+3)    the integer variable
LD (HL),A      load the integer variable
                with the string length
RET            return to BASIC
```

We use a CALL of the form

```
CALL 40000,@A%,@A$
```

to use the routine, having first set up A\$ and set A%=0. Obviously, any string variable and integer variable can be used. Both the below examples are legal uses of the CALL.

```
A%=0:A$="FRED":CALL 40000,@A%,@A$
LE%=0:b$="joe";CALL 40000,@LE%,@b$
```

In both cases, the integer variable will hold the length of the string after the CALL has been executed.

That completes this review of the parameter passing abilities of the Amstrad Computer. As I mentioned, we haven't covered passing real numbers to machine code programs, but I'm sure you'll agree that the facilities offered that we've examined are extremely powerful. Also, they are nice and easy to use, so you've got no excuse for not using them in your programs!

Chapter 6

8 bit counting

In Chapter 4 we examined the ways in which we could transfer data between CPU registers and the computer memory. In the last Chapter we saw how we could use these instructions to write simple machine code programs. However, the programs we saw simply shuffled data around; they didn't do any arithmetic on any of the data they were working on. So, in this Chapter we'll look at the instructions that are available to us for 8 bit arithmetic and counting operations. We'll also look at that "odd man out" amongst the 8 bit registers of the CPU, the F register. In fact, that's where we'll start.

The F Register

The F, or FLAG, register serves to indicate to the CPU that certain events have happened; all of these events are concerned with the arithmetic operations of the CPU, irrespective of whether these are 8 or 16 bit operations. We NEVER treat the F register as just another 8 bit register; in fact there are no data transfer operations that we can do with the F register. We can only PUSH it on to the stack, nothing else.

Instead of treating the F register like a byte, we look at it as 8 separate bits, each bit representing a certain piece of information. These are known as FLAG BITS, or just FLAGS. Although there are 8 bits in the register, only 6 of them actually are used by the CPU. I assume that the Z80 designer ran out of ideas of what to do with the other 2 bits.

There is a little jargon here to do with bits that are used as flags. We say that if a flag bit is set to a value of 1 the flag is SET. If it is set to 0, then the flag is CLEAR or RESET. After an instruction has been executed by the CPU, the flags that have been affected by that command, if any, are updated by the CPU. Not all the commands available to the CPU affect the values of flags; for example, the load instructions that we've already seen don't alter any flags at all.

What the Flags stand for

The below illustration shows how the flag register is arranged.

Bit	7	6	5	4	3	2	1	0
Name	S	Z	X	H	X	P/V	N	C

The significance of each bit will be explained shortly. Two things, though, spring to our immediate notice.

- (a) Bits 3 and 5 are given the name "X". These have no relevance whatsoever to the programming or behaviour of the CPU. "X" is often used in computer circles to designate "Don't care"!
- (b) Bit 2 has a two letter name; this is because it is used as a different flag by different instructions. That is, it means one thing when some instructions are executed and another thing when others are.

The C Flag

This is also known as the CARRY flag. If you think back a little way, you'll remember that 8 bit numbers can be in the range 0 to 255 for 8 bit numbers and 0 to 65535 for 16 bit numbers. Look at the below binary addition;

11111111	Remember that in binary
+00000001	1 + 1 = 0 with a carry of 1
1 00000000	

However, only the 8 zeros are represented in the register; there is no room for the lone 1. Thus, if we were to now inspect the register we'd find the answer 0 instead of the correct answer 256. This is simply because you cannot represent the number 256 on 8 "fingers". When the accuracy of a result is lost in this way we say that we have an OVERFLOW problem. When such a problem is encountered by the CPU, and a "ninth bit" is returned, the "ninth bit" goes into the Carry flag of the F register. A similar problem can occur in subtraction problems, such as 200 - 201. Here the C flag is set to 1 if the subtraction requires the use of a carry from the MSB of the A register. The C flag is thus vitally important in 280 arithmetic operations; it's so important, in fact, that we've been given two commands that enable us to directly manipulate the C flag. These are

SCF with opcode &37
CCF with opcode &3F

SCF stands for Set Carry Flag, and on execution this instruction will set the C flag to 1. CCF stands for

Complement Carry Flag and on execution will set the carry flag to it's opposite state. That is, if the C flag is set to 1 when the instruction is executed it will be altered to 0, and if it is set to 0 when the instruction is executed it will be altered to 1.

The N Flag

This is called the SUBTRACT flag. It is used mainly by the rather special BCD arithmetic instructions that we shall encounter later in this Chapter, and indicates, not surprisingly, that the last instruction was a subtraction when this flag is set!

The H Flag

This is called the HALF CARRY flag, and indicates that a carry or borrow operation has been carried out to or from the 5th bit of a A register. It is used in BCD arithmetic.

The P/V flag

This is the two purpose flag in the flag register. It is called the PARITY/OVERFLOW flag. Let's look at each of it's roles in turn.

Parity

I'm sorry about all these new terms; this is a concept that is involved in what are called the LOGICAL operations of the CPU. We'll look at these in greater detail in a later section. If a byte has an odd number of bits in it set to 1 then the byte is said to have ODD PARITY, and if it has an even number of bits in it set to 1 it is said to have EVEN PARITY. If a byte has odd parity then the P/V flag will be set to 0, otherwise it will be set to 1. As an example of parity,

```
0 1 1 0 1 1 1 1    parity is even
0 0 0 0 0 0 0 1    parity is odd
```

The P/V flag is ONLY used as an indicator of the parity of a byte when logical operations have been performed on the byte.

Overflow

This use of the flag is important when we are dealing with Two's complement arithmetic. It indicates that the addition of two positive numbers represented in two's complement representation has given rise to a NEGATIVE number! This is clearly not possible, and results from, for example, the sum being greater than 127 for 8 bit Two's Complement numbers and 32767 for

16 bit Two's complement numbers. It also indicates when the addition of two negative Two's complement numbers have given rise to a positive number; again not possible. Either of these conditions is signaled by the P/V flag being set to 1.

The Z flag

This is called the ZERO FLAG and is probably the most widely used of all the CPU flags. It indicates whether or not the result of a particular operation was zero. The result being tested should be in the A register, and so the Z flag really tests whether or not an operation left zero in the A register. When the result is zero, the flag is set to 1. When the result is NOT zero, the flag is set to zero. This latter point can cause some confusion!

The S Flag

This is the SIGN FLAG and serves to signify the sign of the result of an operation. It is effectively a copy of the MSB (Most Significant Bit) of the A register, and so in accordance with Two's complement notation will be set to 1 if the result is negative and 0 if the result is positive. This flag, therefore, just gives us the status of bit 7 of the A register.

How are they used?

In BASIC we can have program structures such as

```
IF A=2 THEN ...
```

Well, in machine code programs we can have something very similar. We use the status of different flags, in conjunction with some instructions that we'll encounter later, to form these commands. The only difference is that the instructions after the machine code equivalent of IF ... THEN must be the machine language equivalent of GOTO or GOSUB. The machine code equivalents of these are JP and CALL respectively. DO NOT get this CALL mixed up with the BASIC command CALL that we've already encountered. JP simply stands for Jump, and we'll take a much closer look at these instructions in Chapter 8. For now, we'll say that we use the flags to decide whether or not we make a JP or CALL. For example, the assembler equivalent of

```
IF A=0 THEN GOTO 12000
```

is

```
JP 2,12000
```

This machine code instruction means; if the result of

the last instruction was zero, jump to address 12000 within the computer memory.

This jump instruction is an example of a class of instructions that are called **CONDITIONAL** instructions; these instructions are only executed if a specific condition is met.

However, we're getting ahead of ourselves here; let's look at the instructions that we can use to carry out 8 bit counting and arithmetic operations. Before we leave flags, remember that the effects of various instructions on the flags are indicated on the various Tables of instructions scattered through the book.

Counting with 8 bits

The easiest arithmetic operation that I can think of is simply to add or subtract 1 from a number. The CPU can do this job very easily, and we'll now look at the instructions used to do this.

Counting up

To increase the contents of an 8 bit register by 1 we use the command

```
INC r
```

where *r* is an 8 bit register. Thus a typical instruction might be `INC A`. This will increase the value of the *A* register by 1. Thus if the *A* register was containing 4 before the `INC A` instruction, it would be holding 5 after this instruction. The command

```
INC rr
```

does the same job for the register pairs. More details will be given in Chapter 8 when we discuss the 16 bit instructions. When we are using these 8 bit arithmetic instructions, we can use many of the addressing modes that we looked at in Chapter 4; take the use of Indexed Addressing as an example.

```
INC (IX+d)  
INC (IY+d)
```

are both legal instructions. We specify the address of the memory location of interest in the Index Registers with the displacement byte, and then the execution of the above instructions would lead to the contents of that particular byte of memory being incremented by 1. We can use the *HL* register to specify a memory location, as well. Thus,

```
INC (HL)
```

increments the contents of the memory location specified in the HL register pair. To give an example,

```
LD HL,40000
LD A,0
LD (40000),A
INC (HL)
RET
```

will result in the memory location 40000 containing the value 1 instead of the value 0 with which it was originally loaded. Remember that the brackets indicate that the instruction we're executing refers to the contents of the ADDRESS specified by the register pair, not the register pair itself. Example 4 shows a simple program that uses the INC A instruction to print all the available characters in the Amstrad character set to the screen. In a later Example, we'll see a program in which the actual printing to the screen is also done from machine code!

The assembler language code is;

Example 4:

```
LD L,(IX+0)
LD H,(IX+1)
LD A,(HL)
INC A
LD (HL),A
RET
```

It's very simple; we simply get the address of an integer variable, get the low byte, increment it, put it back and then return to BASIC. We could have used an INC (HL) instruction, that would have replaced 3 of the above instructions. The BASIC program to load the machine code bytes and CALL that above program is shown below. The CALL statement has the same effect as LET A%=A%+1.

```
10 MEMORY 39999
20 A%=31: REM set the variable up
30 FOR I=0 TO 9: REM put the machine code in memory
40 READ A$:POKE (40000+I),VAL("&"+A$)
50 NEXT I
60 :
70 CALL 40000,&A$:REM make the call
80 PRINT CHR$(A%):REM print the character
90 IF A%<256 THEN GOTO 70
100 DATA DD,6E,00,DD,66,01,7E,3C,77,C9
```

The alternative version of the assembler listing given above, but using INC (HL) instruction, is:

```

LD   L,(IX+0)
LD   H,(IX+1)
INC  (HL)
RET

```

You might try to convert these assembler instructions into machine code bytes.

One thing to note about INC instructions is that it is possible to get back to zero by repeatedly incrementing a register or memory location. This isn't as mysterious as it sounds. Remember when we discussed the carry flag? Well, incrementing a register that contains the value 255 will set the lower 8 bits of the register to zero. This will obviously set the contents of an 8 bit register to zero, as we've already seen.

Counting Down

The instructions for counting down with a register or memory location are analogous to those that we've just examined for counting up. So, the below instructions are all legal ones for counting down.

```

DEC  r
DEC  rr
DEC  IX
DEC  IY
DEC  (HL)
DEC  (IX+d)
DEC  (IY+d)

```

Note here the DEC IX and the DEC IY instructions. There are, of course, INC IX and INC IY instructions as well. The DEC instructions all reduce the value held in the register or memory location by 1. DEC can be thought of as standing for DECREASE or DECREMENT. As an example, the below short program will decrement the contents of location 40000 by 1.

```

LD   A,(40000)
DEC  A
LD   (40000),A
RET

```

Just as you can get to zero by repeatedly incrementing a register or memory location, you can achieve a value of 255 by repeatedly decrementing a register. As soon as the register contains the value 0, a further DEC operation will leave the value 255 in the register. This can be quite a useful programming trick under some circumstances.

Let's now go and look at how the INC and DEC instructions affect the CPU flags. These are the first instructions that we've met that actually alter the flags in any way.

Effect on the flags

The 8 bit INC and DEC instructions affect most flags; however, the 16 bit register pair INC and DEC instructions DO NOT affect any of the flags. This can be extremely frustrating, as it means that we have to use extra instructions to see if the register pair has reached 0. Whether this omission was an oversight on the part of the CPU designer, or an attempt to undermine the sanity of programmers, we'll never know. The only important flag that is NOT affected by the 8 bit INC and DEC instructions is the C flag. The notes given below, therefore, only refer to the 8 bit instructions.

Sign Flag

This is set if bit 7 of the result is set to 1.

Zero Flag

This is set if the value of the result is zero.

Overflow

This flag is set if the operation altered the value of bit 7 of the result.

Half Carry

This is set if there is a carry or borrow from bit 4 of the result.

Negate

This is set if the last instruction was a subtraction instruction, and so is set if a DEC instruction was the last instruction executed, as DEC is simply a special form of the subtraction operation.

Well, the ability to count up and down is rather useful; it enables us to simulate things like the BASIC FOR ... NEXT loops, as we shall see when we look at jumps and calls in a later Chapter. However, they are of little use to us if we want to add two numbers together, such as $56+32$. For this, we need some more sophisticated arithmetic instructions, and we'll look at these next.

8 bit Arithmetic

In this section we'll consider the 8 bit arithmetic operations, addition and subtraction. Both these types of operation involve the A register; indeed, there are some instructions that take the A register so much for granted that it is not even mentioned in the mnemonic

of the instruction! The arithmetic operations available to the Z80, you will have noticed, do not include the operations of multiplication and division.

Some CPU's do include these two operations in the instruction set, but the Z80 doesn't. If we want to do either of these operations in machine code then we have to write short machine code programs to do so.

The simplest arithmetic operation is the 8 bit addition, so let's start there.

8 Bit addition

The simplest 8 bit add operation is

```
ADD A,r
```

where *r* is one of the 8 bit registers. This instruction tells the CPU to add the contents of the register *r* to the contents of the A register and leave the result of the addition in the A register. This is the historical origin of the full name of the A register, which is the ACCUMULATOR. In some of the early computers, one particular register within the computer was used to "accumulate" the results of various operations done by the computer. That is exactly the role of the A register here. As an example of the use of these addition instructions, the below instruction adds the contents of the A register to the B register.

```
ADD A,B
```

All the 8 bit addition operations can affect the values of ALL the CPU flags; they set the N flag to 0, and alter the other flags to reflect the result of the operation just performed. The 8 bit additions are available in a wide range of addressing modes:

```
ADD A,r
ADD A,n
ADD A,(HL)
ADD A,(IX+d)
ADD A,(IY+d)
```

The one addressing mode that we've not got above is the Extended Addressing mode i.e.

```
ADD A,(nn)
```

where *nn* is a 16 bit address. However, this is no real problem, as we can simulate this operation by

```
LD HL,nn
ADD A,(HL)
```

Don't worry about the 16 bit load instruction; this

simply puts the value nn into the HL register pair. We'll examine these instructions in Chapter 7.

A very important thing to remember is that the result of an 8 bit addition operation must be capable of being represented as a 8 bit number. For example,

```
LD   A,128
ADD  A,128
```

will leave the value 0 in the A register, and not the true result of this addition, which is 256. The reason for this is quite obvious; you can't represent 256 on 8 bits.

However, this type of addition WILL set the carry flag, and so we'll be aware of the error that has occurred. Let's now look at a simple addition program that adds two integer variables together and passes the result back to BASIC. Because we haven't yet looked at 16 bit data transfers or 16 bit addition operations, the numbers used should be such that the result of the operation will be less than 255. However, if you want to be awkward, and enter bigger numbers, you'll get an appreciation of the overflow problems!

The assembler listing for this program is shown below;

Example 5:

```
LD   L,(IX+0)
LD   H,(IX+1)
LD   A,(HL)
LD   L,(IX+2)
LD   H,(IX+3)
ADD  A,(HL)
LD   (HL),A
RET
```

the instructions above leave the result in the A% variable for the CALL shown in the below BASIC program. The below BASIC program will POKE the bytes that make the above program into memory and CALL it.

```
10 MEMORY 39999
20 FOR I=0 TO 15
30 READ A#;POKE (40000+I),VAL("&" +A#)
40 NEXT I
50 :
60 INPUT "First number",A%
70 INPUT "Second number",B%
80 CALL 4000,@A%,@B%
85 PRINT A%
90 END
100 DATA DD,6E,00,DD,66,01,7E,DD,6E,02,DD,66,03,86
    77,C9
```

This will print the sum of the two variables to the

screen, assuming the variables give a sum in the range 0 to 255. Note that it can't cope with negative numbers.

If you play about with the above routine, and type in something like 123+245, you'll get the result 112 instead of the real answer 368. This is the overflow problem coming to the surface.

$$112 = 368 - 256$$

What can we do to get around this problem? Well, the answer is to use an instruction called ADC. ADC stands for Add with Carry, and enables us to take the carry generated in these conditions and do something useful with it.

Add with Carry

These instructions work in the same way as the normal ADD instructions, have the same addressing modes and work on 8 bit numbers. Where they differ is that when we use ADC, the value of the Carry flag, whether 1 or 0, is added to the result generated. It thus enables us to make use of the carry generated. Let's take a look at an example to make it clearer. Let's carry out an addition that generates a result that can only be accurately represented in 16 bits, such as $100+200=300$.

The below machine code program would, if entered at a suitable address in the machine, put the result of the addition into addresses 41000 and 41001. The Low byte of the result will be placed in location 41000 and the High byte of the result will be placed in location 41001.

LD	A,200	set up A and B with the
LD	B,100	numbers to be added
ADD	A,B	add them, A is low byte of sum
LD	(41000),A	store the low byte
LD	A,0	the high bytes of the numbers
LD	B,0	being added are zero
ADC	A,B	add high bytes with the carry
		in this case $A = 0 + 0 + 1$
LD	(41001),A	store the high byte
RET		return to BASIC

If you like, hand assemble this program and enter it into your computer at address 40000. Remember that all addresses in the program should be entered into the memory low byte first. Use a CALL 40000 instruction from BASIC to run the program. The contents of the two locations holding the result can then be PEEKed out using

```
PRINT PEEK(41000)+256*PEEK(41001)
```

Thus, we can do arithmetic with 16 bit numbers using only 8 bit arithmetic instructions. Thus, we could use a program similar to that above to add any 16 bit numbers together. One point to remember is that the Amstrad BASIC Interpreter uses the Two's Complement notation for it's 16 bit integers, and so if we were to use the parameter block to pass such integers back to BASIC, any number above +32767 in value would be passed back to BASIC as a negative number. If we want to get around this problem, then we must store the result that we want passing back to BASIC in a couple of memory locations, like we did above, and then access these locations using PEEK instructions. This would allow us to return large 16 bit numbers to BASIC quite easily.

Example 6:

As a final example of 8 bit addition instructions, Example 6 shows a general routine to add together two integer variables and pass the result back in an integer variable. This routine will handle negative numbers.

The CALL used by this routine is of the form

```
CALL 40000,A%,B%,@C%
```

the values to be added are passed over in A% and B% and the answer is returned in C%. For this reason, we have to use the @ symbol to pass parameter C%. It also means that we must set C% to some value before calling the machine code. The instruction

```
CALL 40000,2,4,@C%
```

is equally valid. The assembler listing is given below.

```
LD L,(IX+0)    get the address of C%
LD H,(IX+1)
LD A,(IX+2)    low byte of B% into A
LD B,(IX+4)    low byte of A% into B
ADD A,B        form the sum in A
LD (HL),A     put result in low byte of C%
INC HL        make HL point to high byte of C%
LD A,(IX+3)    get the High byte of B%
LD B,(IX+5)    get the High byte of A%
ADC A,B        add with carry to get high byte
LD (HL),A     store in high byte of C%
RET           return to BASIC
```

Enter the program at address 40000 after assembling it. After a call the C% variable will hold the result of the addition. Try typing in negative numbers, and see how the computer converts large integers into negative numbers on the return to BASIC by typing in something like 32767+5. You can, of course, use

different variable names in the above. However, remember that the last parameter passed should be a variable which is prefixed by the "@" symbol. Finally, remember that the last parameter should be initialised in some way, even if it's just set to zero.

8 bit subtraction

The instructions for this operation are analogous to those for 8 bit additions. Again, there are two different types of subtraction operation. These are

- SUB subtract
- SBC subtract with carry

I won't go into too much detail here about these instructions, because of their overall similarity to the 8 bit addition instructions. The result of any 8 bit subtraction is left in the A register. The SBC instruction subtracts the value of the C flag as well as the value of the operand.

Before leaving these operations, a couple of general points. Note that both ADD A,A and Sub A,A commands exist. These will have the result of doubling the A register contents and setting them to zero respectively. The SUB A,A command is also a good method of clearing the C flag. It is important in the ADC and SBC operations that you keep a close eye on the C flag.

The final point is one of mnemonics. Certain of the instructions we've just examined are occasionally written without mentioning the A register. Thus,

- ADD (IX+d) means ADD A,(IX+d)
- ADD B means ADD A,B
- ADD 23 means ADD A,23

So, if you ever come across any of these ways of writing instructions down, you'll know what is meant.

BCD Arithmetic

So far, all the arithmetic that we've dealt with has been binary.

In BCD, or Binary Coded Decimal arithmetic we treat the 8 bit byte that we know and love in a totally new way. However, don't worry too much, as it's only included here for the sake of completeness. The byte is now treated as two 4 bit nibbles, each nibble representing a decimal digit between 0 and 9. The first point to note about this is that 6 of the available 16 4 bit combinations are not used. This is therefore an inefficient form of representing numbers, but is often useful when the CPU has to communicate with certain types of peripheral device. To see the

difference between BCD and straight binary representation of numbers, look at the below examples.

0010 0010 = 22 in BCD or 34 in binary.
1001 1001 = 99 in BCD or 153 in binary.
 This is the largest 8 bit BCD number.
1101 1101 = is 221 in binary,
 but is ILLEGAL in BCD.

Arithmetic with BCD numbers

This peculiar representation method can cause a pot full of problems when we start doing addition and subtraction with numbers. Consider the addition below.

BCD 08 is 0000 1000
BCD 03 is 0000 0011
Their sum is 0000 1011

This is the correct answer in binary, but it is illegal in BCD representation. We must somehow convert the binary result of such an operation into a legal BCD number. We can get a proper BCD number by adding 6 to the number PROVIDED THAT the low nibble has a value of greater than 9. Fortunately, we don't have to work out a program to do this ourselves; the designers of the CPU put in an instruction to convert any illegal BCD byte into a legal BCD byte. The instruction is called DAA, which stands for Decimal Adjust Accumulator. any "carry" from the low nibble to the high nibble is indicated by our old friend, the H flag.

Having said all that, it's not likely that you'll use BCD in your programs very often!

Comparing Numbers

You might ask what a section on comparing numbers is doing in a Chapter on 8 bit counting operations. Well, the process of comparing two numbers, as far as the CPU is concerned, is essentially one of subtraction. To work out which of two numbers is the larger, we carry out a subtraction operation, and examine the result. The result will be either 0, positive or negative, and so will indicate the relative magnitude of the two numbers involved.

The Z80 CPU has a special instruction to perform this operation, which is not surprisingly called the Compare operation. The instruction is CP.

The CP Instruction

The CP instruction operates in a variety of addressing modes, but always compares the contents of the A register with either the contents of another register,

memory location or number. The addressing modes available are;

```
CP r
CP n
CP (HL)
CP (IX+d)
CP (IY+d)
```

If we look at the instruction CP r, then the operation is effectively a subtraction of the contents of the specified register from the contents of the A register. However, the result is NOT placed in the A register. Although the A register is not mentioned in the above instructions, the CP instructions always use it. It should be fairly obvious that the CP instructions alter the flags; if they didn't there'd be no way of telling what the result of a comparison was! The N flag is set to indicate a subtraction operation. The other flags are altered depending upon the result of the operation. The P/V flag is used as an overflow flag.

The two flags that are most commonly used when we're comparing numbers are the C and Z flags. Look at the following example;

```
LD A,10
CP B
```

The Z and C flags will be set in the below fashion.

```
If B < 10 then Z = 0 and C = 0
If B = 10 then Z = 1 and C = 0
If B > 10 then Z = 0 and C = 1
```

We thus have three unique sets of flags that we can use to make decisions with. The status of these flags can be used by conditional jump or call instructions to control the flow of the program. A couple of other examples of the CP instruction's effect on flags are shown below.

```
LD A,23
LD B,22
CP B
```

will have both Z and C set to zero.

On the other hand,

```
LD A,20
LD B,30
CP B
```

will leave Z set to zero and C set to 1.

The main point to note about the CP instruction in all it's forms is that it is only by testing the status of the flags that we can tell the result of the

comparison operation. Thus the flag test must be carried out before any commands are executed that alter the flags again.

Mnemonic	Bytes	Time Taken	Effect on Flags					
			C	Z	P/V	S	N	H
ADD A,register	1	4	#	#	#	#	0	#
ADD A,number	2	7	#	#	#	#	0	#
ADD A,(HL)	1	7	#	#	#	#	0	#
ADD A,(IX+d)	3	19	#	#	#	#	0	#
ADD A,(IY+d)	3	19	#	#	#	#	0	#
ADC A,register	1	4	#	#	#	#	0	#
ADC A,number	2	7	#	#	#	#	0	#
ADC A,(HL)	1	7	#	#	#	#	0	#
ADC (IX+d)	3	19	#	#	#	#	0	#
ADC A,(IY+d)	3	19	#	#	#	#	0	#
SUB register	1	4	#	#	#	#	1	#
SUB number	2	7	#	#	#	#	1	#
SUB (HL)	1	7	#	#	#	#	1	#
SUB (IX+d)	3	19	#	#	#	#	1	#
SUB (IY+d)	3	19	#	#	#	#	1	#
SBC A,register	1	4	#	#	#	#	1	#
SBC A,number	2	7	#	#	#	#	1	#
SBC A,(HL)	1	7	#	#	#	#	1	#
SBC A,(IX+d)	3	19	#	#	#	#	1	#
SBC A,(IY+d)	3	19	#	#	#	#	1	#
CP register	1	4	#	#	#	#	1	#
CP number	2	7	#	#	#	#	1	#
CP (HL)	1	7	#	#	#	#	1	#
CP (IX+d)	3	19	#	#	#	#	1	#
CP (IY+d)	3	19	#	#	#	#	1	#

Flags Notation:

indicates flag is altered by operation
0 indicates flag is set to 0
1 indicates flag is set to 1
- indicates flag is unaffected

Table 2. 8 bit arithmetic and comparisons

Logical Operations

Strictly speaking, of course, all the operations of a computer are logical! However, the operations to which I'm referring here are operations on the bits within 8 bit bytes, rather than operations on the bytes as a whole. The instructions in this category can be as valuable to the programmer in machine code as the operations of addition and subtraction. These bit oriented operations are called **BOOLEAN OPERATIONS**,

after the Dublin mathematics Professor who first described them. Just as addition and subtraction are said to be arithmetic operators, the instructions that we'll look at in this section are called Logical, or Boolean, Operators.

There are 4 Boolean Operators that are supported by the Z80 instruction set. These are

XOR
AND
NOT
OR

Let's look at each of these and see what use we can put them to in our programs.

Truth Tables

A quick diversion; just as we have tables to describe the rules of multiplication, we have tables to tell us what the result of a given logical operation will be. These are called TRUTH TABLES. Each of the separate operations listed above has a different Truth Table, as we shall now see.

The NOT Operation

The action of the NOT operator is shown below; it works on one bit.

A	NOT A
0	1
1	0

You will note that this is a COMPLEMENT operation, like the one that we saw when we discussed the Two's complement method of representing numbers. The instruction to complement the A register is CPL. This is the only register that you can actually complement, and no other addressing modes are supported. One application of this instruction is in the generation of the Two's complement of a number. The below instructions will do this.

CPL
INC A

The AND Operation

There are 11 different AND instructions available to the programmer in the Z80 instruction set. They operate in the below addressing modes.

```

AND r
AND n
AND (HL)
AND (IX+d)
AND (IY+d)

```

Note how the A register is not mentioned at all here, despite the fact that all these operations will only work with this register as one of those involved. The AND operation works on one bit from each register, and alters all the bits in the A register according to the corresponding bits in the other register.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

It is clear from this that the result of an AND operation is to set a bit to 1 only if both the bits being ANDed together are 1. The command

```
AND B
```

will do an AND operation on the A and B registers. The result of such an operation on a particular set of data is shown below.

A	00000101
B	00010000
A AND B	00000000

Alternatively, we might have,

A	00000101
B	00001111
A AND B	00000101

You can see that we can use the AND instruction to "mask off" certain bits of the byte held in the A register in which we ensure that a byte only has a value between 0 and 15 inclusive. This is simple. We just ignore the high nibble of the byte, and just disregard bits 4 to 7 of the accumulator as shown below.

```

LD A,(40000)
LD B,15          15 is binary 00001111
AND B
LD (40000),A

```

Thus no matter what value is held in the upper 4 bits of the A register, it will be ANDed with zero, giving a result of zero for the upper 4 bits. This leads us rather neatly on to another use of the AND command; the setting of certain bits in a byte to zero without

affecting other bits in the byte. The particular bit that we want to set to zero is simply ANDed with zero. For example, to set bit 5 to zero;

```
LD B,223      223 is 11101111 in binary
AND B
```

The OR Operation

This instruction has the truth table shown below.

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

The addressing modes that are available to the OR instruction are as for the AND instruction. The Z80 only allows us to OR something with the contents of the A register. We can use this command to set given bits in the A register or another register or memory location to 1. For example, to set bit 1 of the A register to 1 we could use

```
LD B,2
OR B
```

or, more simply,

```
OR 2
```

There are other instructions that we can use to set specific bits in this way, as we shall soon see.

The ASCII code of the lower case letters is 32 higher than the ASCII code of the corresponding upper case letter. The below machine code program accepts a string from a CALL statement, then returns it to BASIC having first converted the first letter of the string to lower case. We use the OR instruction to set bit 5, and hence effectively add 32 to the ASCII code of the character. The assembler code is given below.

Example 7:

```
LD L,(IX+0)
LD H,(IX+1)
INC HL
LD C,(HL)
INC HL
LD B,(HL)
LD A,(BC)
OR &20
LD (BC),A
RET
```

Assemble the program into memory at address 40000. You can then CALL it using

```
CALL 40000,@S$
```

where S\$ is the string of interest. One interesting point to note here occurs in the below program section.

```
100 S$="FRDD"  
110 CALL 40000,@S$
```

After executing it, the command PRINT S\$ will return "fRDD". However, line 100 is also altered, as this is where the string is defined in memory. Line 100 will now read

```
100 S$="fRDD"
```

This is something to be aware of.

The XOR Operation.

Well, contrary to popular belief, XOR is not a character from a Science Fiction novel! It is an abbreviation of the name EXCLUSIVE OR. The truth table for this operation is shown below.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

As you can see, the XOR instruction gives a 1 as a result only when the two bits being operated on have different values. The addressing modes that are available are the same as those that are available for the AND instruction. One use of the XOR instruction is to set the A register to zero;

```
XOR A
```

will clear A to zero and only requires one byte to do so.

Effect on Flags

As can be seen from the Table at the end of this section, all the flags are affected in some way by these commands. However, only three of the flags have a status that depends upon the outcome of such operations. These are as follows:

- Z This will be set if the result is zero.
- S This will be set to 1 if bit 7 of the result is 1.

P/V This acts as a Parity Flag, and will be set for even parity and clear for odd parity.

We can use the Boolean Operators to manipulate the C flag status in the below fashion;

OR A clears C and leaves A alone
XOR A clears both C and A register

Mnemonic	Bytes	Time Taken	Effect on Flags						
			C	Z	P/V	S	N	H	
AND register	1	4	0	#	#	#	0	1	
AND number	2	7	0	#	#	#	0	1	
AND (HL)	1	7	0	#	#	#	0	1	
AND (IX+d)	3	19	0	#	#	#	0	1	
AND (IY+d)	3	19	0	#	#	#	0	1	
OR register	1	4	0	#	#	#	0	0	
OR number	2	7	0	#	#	#	0	0	
OR (HL)	1	7	0	#	#	#	0	0	
OR (IX+d)	3	19	0	#	#	#	0	0	
OR (IY+d)	3	19	0	#	#	#	0	0	
XOR register	1	4	0	#	#	#	0	0	
XOR number	2	7	0	#	#	#	0	0	
XOR (HL)	1	7	0	#	#	#	0	0	
XOR (IX+d)	3	19	0	#	#	#	0	0	
XOR (IY+d)	3	19	0	#	#	#	0	0	

Flags Notation:

indicates flag is altered by operation
0 indicates flag is set to 0
1 indicates flag is set to 1
- indicates flag is unaffected

Table 3. Logical Operations

Manipulating Bits in a Byte

There are a couple of commands in the Z80 instruction set that enable us to manipulate the status of bits within a memory location or register. These instructions are called SET and RESET, and much of what you can do with them can also be done with the logical operations seen above.

The SET command sets a given bit to 1. It operates in the below addressing modes;

SET n,r
SET n,(HL)
SET n,(IX+d)
SET n,(IY+d)

where n is the bit to be set. n has a value between 0 and 7. Thus the commands

```
LD  A,0
SET 0,A
```

will result in the A register holding the value 1. No changes are made to the flags. The complementary command to SET is RESET, and this will force the value of a particular bit to zero. It operates in the same addressing modes as SET. Thus the command

```
LD  A,255
RES 0,A
```

will result in the A register holding the value 254.

It is possible to test the value of an individual bit within a byte using the BIT instruction. It functions in the same addressing modes as SET and RESET, and signals it's result via the Z flag. Look at the following example;

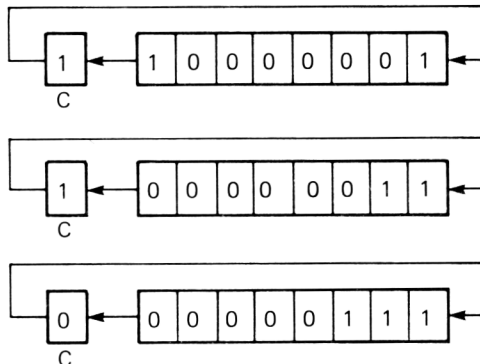
```
LD  IX,200
BIT 0,(IX+0)
```

This will test bit 0 of location 200. If this bit was equal to zero, then the Z flag will be set to 1. Otherwise, it's not.

The final group of commands that we 'll look at in this Chapter are the ROTATE and SHIFT instructions. These aren't often used in machine code programming, but on the occasions that they are used they are very valuable indeed.

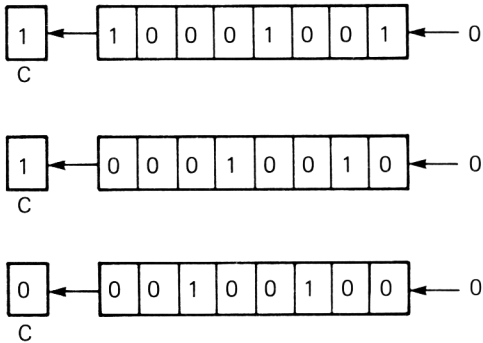
Rotates and Shifts

On the whole, a rotate operation moves bits within a byte in such a way that the bits moved out of one end of a byte are eventually moved into the other end of the byte. This cyclic operation is best shown as



Shifts, on the other hand, are non-cyclic

operations which result in bits being "lost", as shown below.



A Rotate or Shift operation is named according to the direction the bits are shifted in. If the bits are shifted to the left then it is a ROTATE or SHIFT LEFT. If the movement is to the right it is a SHIFT RIGHT.

Left Operations

There are two different Rotate Left operations and one Shift Left operation.

Rotate Left

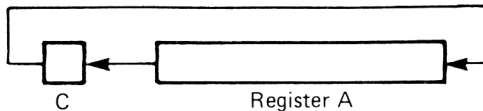
This operation can be applied to the below addressing modes.

- RL r
- RL (HL)
- RL (IX+d)
- RL (IY+d)

A typical example is

```
RL A
```

which stands for Rotate Left A register. The action of the command is shown below.



The current value of the C flag goes into the A register into bit 0. Bit 7 of the register enters in to the C flag. In this case, the C flag can be looked at as a "9th bit".

Rotate Left Circular

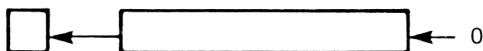
This operates in the same addressing modes that were featured above. The mnemonic is RLC. For example, typical commands are RLC A and RLC (HL). The difference between this operation and the last one is that the value of the C flag is not cycled into bit 0 of the register involved.

Shift Left Accumulator

This operates on the same addressing modes as above. The mnemonic is

SLA s

where s represents any of the addressing modes. It's operation can be seen below;



This can effectively be seen as a "multiply by 2" instruction. However, if the value in the register is greater than 127 when the SLA is executed, a "funny" result will be returned. You have been warned!

Example 8:

This example shows SLA A at work. Enter the code at address 40000. CALL 40000,@A% will return A%/2. A% must be initialised to start with, and should be within the range 0 to 127.

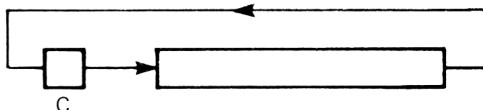
```
LD L,(IX+0)
LD H,(IX+1)
SLA (HL)
RET
```

Right Operations

The simplest instructions are the Rotate Right instructions. These instructions are;

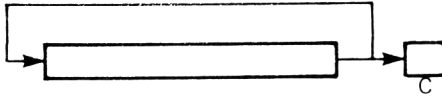
```
RR r
RR (HL)
RR (IX+d)
RR (IY+d)
```

The operation can be seen as



Rotate Right Circular

The same addressing modes are supported as for the RR instructions and the operation is best seen as

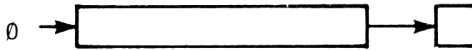


Shift Right Logical

The addressing modes available for this instruction are

```
SRL r
SRL (HL)
SRL (IX+d)
SRL (IY+d)
```

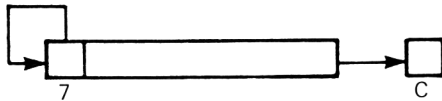
and the operation can be looked at as a divide by two, bit 7 being set to zero and bit 0 being pushed into the carry.



One problem with this instruction when used as a divide by two operation is where we have a number that is a two's complement number. It is vital in this event that the sign bit, bit 7, is retained. There is a command to do this.

Shift Right Arithmetic

This retains the current status of bit 7, but in all other respects is the same as SRL. The mnemonic is SRA.



Those of you who are still with me will no doubt be gratified to know two things;

- (a) That 's the end of the 8 bit arithmetic and logic operations.
- (b) There aren't as many 16 bit arithmetic and logic operations!

In the next two Chapters we'll examine the 16 bit data transfer operations and logic operations.

Chapter 7

16 bit transfers

We've already seen how some of the 8 bit registers can be paired together to form 16 bit register pairs. These offer us the potential for handling 16 bit numbers, and in this Chapter we'll look at how we can transfer 16 bit numbers between CPU register pairs and memory. We've already encountered one use of 16 bit numbers in machine code programming; that of specifying an address within the computer memory.

Let's start with a look at the instructions for loading a 16 bit register pair with a 16 bit number. The general mnemonic for this type of instruction is

```
LD rr,nn
```

where rr is a 16 bit register pair, such as BC, DE or HL. nn is a 16 bit number. A typical example of these instructions is

```
LD HL,&9C00
```

Remember that when we specify the 16 bit number in the instruction, it must be stored with it's low byte first. Thus the above assembler instruction would be coded as

```
LD HL,&9C00
```

which assembles to

```
21 00 9C
```

We can also load numbers directly into the IX and IY registers, such as

```
LD IY,&9C00
```

or

```
LD IX,&9C00
```

Transfers between register pairs and memory

It's all very well to be able to load a register pair with a number, but what about transferring the number to memory? The instructions that can be used for these operations are

```
LD (nn),rr
LD (nn),IX
LD (nn),IY
```

The instructions for loading the 16 bit register pairs with the contents of an address are as follows;

```
LD rr,(nn)
LD IX,(nn)
LD IY,(nn)
```

Because we are dealing with 16 bit numbers, we are actually loading the register pair from address nn and address nn+1. However, we don't have to specify this to the CPU, it takes it into account automatically. In the example,

```
LD HL,(&9C00)
```

will load the HL register pair from addresses &9C00 and &9C01. The L register will receive the contents of address &9C00 and the H register will receive the contents of address &9C01. A possible application of these instructions is shown below;

```
LD HL,(&9C00)
LD A,(HL)
```

In a similar fashion,

```
LD BC,(&9C00)
```

will load the BC register from addresses &9C00 and &9C01. One thing to note is that a command such as

```
LD DE,&9C00
```

will load the value 00 into the E register and the value &9C into the D register. Some of the 16 bit load instructions have 4 byte opcodes - two bytes representing the instruction and two bytes the address of interest.

Mnemonic	Bytes	Time Taken	Effect on Flags					
			C	Z	P/V	S	N	H
LD reg_pair,number	3/4	10	-	-	-	-	-	-
LD IX,number	4	14	-	-	-	-	-	-
LD IY,number	4	14	-	-	-	-	-	-
LD (address),BC	4	20	-	-	-	-	-	-
LD (address),DE	4	20	-	-	-	-	-	-
LD (address),HL	3	16	-	-	-	-	-	-
LD (address),IX	4	20	-	-	-	-	-	-
LD (address),IY	4	20	-	-	-	-	-	-
LD BC,(address)	4	20	-	-	-	-	-	-
LD DE,(address)	4	20	-	-	-	-	-	-
LD HL,(address)	3	16	-	-	-	-	-	-
LD IX,(address)	4	20	-	-	-	-	-	-
LD IY,(address)	4	20	-	-	-	-	-	-

Flags Notation:

indicates flag is altered by operation
0 indicates flag is set to 0
1 indicates flag is set to 1
- indicates flag is unaffected

Table 4. 16 bit data transfer instructions

Manipulating the Stack

Some time ago we looked briefly at the stack; you may recall that this is where the CPU can store items of data without the need to keep a record of where the information was put. We too can use the stack to store information in this way; however, we are limited to storing 16 bit numbers on the stack. The instructions for storing registers on the stack are as follows;

```
PUSH AF
PUSH BD
PUSH DE
PUSH HL
PUSH IX
PUSH IY
```

Note how we treat the AF register pair here like the other register pairs. All these PUSH operations have single byte opcodes. The PUSH commands copy the contents of the relevant register pair on to the stack. The register pair involved still retains a copy of the number that has been pushed onto the stack. To recover a number from the stack, and put it into a register, we use an instruction called POP. This

removes the last item from the stack and puts it into the register involved. Thus

```
POP HL
```

will put the last item on the stack into the HL register. We can use the stack to implement data transfers between the 16 bit registers in the below fashion. The instruction

```
LD BC,DE
```

is not implemented in the Z80 instruction set, and we normally carry out this operation by the instructions

```
LD B,D
LD C,E
```

The operations

```
PUSH DE
POP BC
```

have the same effect.

The instructions also enable us to look directly at the contents of the F register;

```
PUSH AF
POP BC
```

The BC register now contains the original contents of the AF register "pair". The C register will contain the contents of F and the B register will contain the contents of A. The stack can also be used to store the current status of the F register while we do other jobs. e.g.

```
PUSH AF
...      other instructions
POP AF
```

Important Note

The BASIC Interpreter of the Amstrad uses the stack, just as we do. When we execute CALL from BASIC, the address from which the CALL was made, and to which the CPU must return after executing your machine code program, is stored on the stack. As the CPU expects to find this when your machine code has been executed, it is VITAL that this address is the next one available on the stack when the final RET is made in your program to return to BASIC. This RET causes the CPU to jump back to the address that is currently available on the stack, and so obviously if the number on the stack is not the address from which the CALL was made, you're in BIG trouble! So, all PUSHes that you make in your program should be matched by POPs. For example, a program that just consisted of

```
PUSH BC
RET
```

would cause problems, where

```
PUSH BC
POP  BC
RET
```

would be alright, because the POP is balanced by a PUSH.

Moving the Stack

Although most of the time we don't need to know anything about where the stack is located in the computer memory, the CPU, whilst under the control of the Amstrad Operating System, has to set up the stack as one of it's first tasks. It does this by loading the address at which it wishes the stack to be placed into a 16 bit register pair called the STACK POINTER, or SP. The commands that are available to manipulate the stack are as follows;

```
LD   SP,nn
LD   SP,(nn)
LD   SP,IX
LD   SP,IY
```

Obviously, moving the stack around in memory is not an advisable activity until you've gained experience, and you should ALWAYS set SP back to it's original value before going back to BASIC!

The final group of 16 bit data transfer operations that we'll look at are those that transfer data between the main register set and the Alternate Register set. On the whole, it's not advisable to do this on the Amstrad, because the alternate registers are used by the Operating System for various things. Thus, I'll mention the commands, but do not advocate using them!

The command

```
EX   AF,AF'
```

swaps the contents of the AF register with the contents of the AF' register. The instruction

```
EXX
```

swaps the BC, DE and HL register pairs simultaneously with their counterparts. The final exchange instruction doesn't work with the Alternative register set but on two register pairs from the main register set;

EX DE,HL

will swap the contents of the DE and HL registers.

The below table summarises the stack operations.

Mnemonic	Byte	Time Taken	Effect on flags					
			C	Z	P/V	S	N	H
PUSH reg_pair	1	11	-	-	-	-	-	-
PUSH IX	2	15	-	-	-	-	-	-
PUSH IY	2	15	-	-	-	-	-	-
POP reg_pair	1	10	-	-	-	-	-	-
POP IX	2	14	-	-	-	-	-	-
POP IY	2	14	-	-	-	-	-	-
LD SP,address	3	10	-	-	-	-	-	-
LD SP,(address)	3	20	-	-	-	-	-	-
LD SP,HL	1	6	-	-	-	-	-	-
LD SP,IX	2	10	-	-	-	-	-	-
LD SP,IY	2	10	-	-	-	-	-	-

Flags Notation:

indicates flag is altered by operation
0 indicates flag is set to 0
1 indicates flag is set to 1
- indicates flag is unaffected

Table 5. Stack manipulation instructions

Before leaving this Chapter, let's look at an example of a 16 bit data transfer operation. This routine will give you the value of the address from which the CALL command was made in the BASIC Interpreter.

Enter the code at address 40000, and CALL it with the line

Example 9:

```
A%=0:CALL 40000,@A%
```

A% will then hold the address of interest, which will be the address that is RETURNed too by the RET command at the end of the program.

```
POP BC          get the address
PUSH BC         restack the result
LD L,(IX+0)
LD H,(IX+1)
LD (HL),C
INC HL
LD (HL),B
RET
```

In the next Chapter we'll examine the 280 instructions for 16 bit arithmetic operations.

Chapter 8

16 bit arithmetic and counting

The 16 bit arithmetic operations were included in the instruction set of the Z80 to make 16 bit additions and subtractions more convenient. We've already seen how we can make do with the ADC operation for 16 bit addition. The 16 bit arithmetic operations are not as versatile as the 8 bit instructions.

Let's start by examining the 16 bit INC and DEC instructions, as these are the simplest form of arithmetic operation.

Increment and Decrement

The simplest 16 bit INC operation is of the form;

```
INC rr
```

which increments the contents of one of the 16 bit register pairs. Thus the two instructions;

```
LD HL,0000
INC HL
```

will result in the HL register holding the value 1. We can also alter the value held in the Index Registers:

```
INC IX
INC IY
```

The DEC instructions are analogous to the INC instructions. The 16 bit DEC instructions are thus as follows.

```
DEC BC
DEC DE
DEC HL
DEC IX
DEC IY
```

The fundamental difference between the 8 bit and the 16 bit INC and DEC instructions is that the 16 bit operations DO NOT affect any of the flags! Thus, we have to perform extra programming operations to test the value in a 16 bit register after an INC or DEC

instruction. For example, to test if a register pair is equal to zero we have to perform a series of operations like those shown below.

```
DEC HL
LD A,H
OR H
JP Z,address
```

Here, we use the JP Z instruction to pass control to another part of the program if the result of the OR operation is zero. This will only be so IF the H and L registers are both equal to zero. If this is so, then the OR operation will set the zero flag.

Addition and Subtraction

In the same way that the A register is the favourite register for 8 bit addition and subtraction operations, the HL register pair is the favoured register pair for the 16 bit addition and subtraction operations. The ADD instructions that work

```
ADD HL,rr
ADD IX,BC
ADD IX,DE
ADD IX,SP
ADD IX,IX
ADD HL,SP
ADD IY,BC
ADD IY,DE
ADD IY,SP
```

You'll notice a couple of things from this list; the first is that there is no instruction for adding the contents of the HL pair to either of the Index Registers. The second is that no instructions are available for the

```
ADD IX,IY
```

instruction. Finally, note that there is no instruction for

```
ADD HL,nn
```

To perform this job, we must use something like the below method, which has the disadvantage of using two registers.

```
LD DE,nn
ADD HL,DE
```

In all these operations, the result is left in the first register pair to be mentioned in the instruction. Thus in the operation

```
ADD HL,BC
```


the result is left in the HL register pair.

Effect on the Flags

There aren't many flags bothered by the 16 bit arithmetic operations; the C flag is set if there's a carry from bit 7 of the upper register to the "17th bit". Any carry from the low register is automatically carried into the upper register by the addition operation. The only other flag that is affected is the N flag which is set to zero.

There are no 16 bit SUB instructions. Any 16 bit subtractions that we want to do have to be Subtract with Carry operations.

Add and Subtract with Carry

We have a selection of ADC instructions which, as in the case of the 8 bit ADC instructions, give us the opportunity to perform multi-byte addition. The ADC instructions available are:

```
ADC HL,BC
ADC HL,DE
ADC HL,HL
ADC HL,SP
```

You will note that there are no instructions to deal with the IX or IY registers here.

The SBC instructions are analogous to the above ADC instructions. They are:

```
SBC HL,BC
SBC HL,DE
SBC HL,HL
SBC HL,SP
```

Because of the fact that all these operations involve the C flag in their reckonings, remember to clear the C flag before doing any subtractions that do not need to SBC instruction's added complexity. How can we do this? Well, the easiest way is to use a Boolean operation to clear the C flag. The below instructions will perform a simple 16 bit subtraction, subtracting the contents of locations 41002 and 41003 from the contents of locations 41000 and 41001. The result ends up in the HL register and is stored in locations 41000 and 41001 before return to BASIC.

```

LD    HL,(41000)
LD    DE,(41002)
AND   A
SBC   HL,DE
LD    HL,DE
LD    (41000),HL
RET

```

The below Table shows the effects on the flags of these instructions. The Z, P/V and S flags have a status that depends upon the result of the operation.

Mnemonic	Bytes	Time Taken	Effect on Flags					
			C	Z	P/V	S	N	H
ADD HL,reg_pair	1	11	#	-	-	-	0	?
ADD HL,SP	2	11	#	-	-	-	0	?
ADD HL,reg_pair	2	15	#	#	#	#	0	?
ADD IX,SP	2	15	#	#	#	#	0	?
ADD IX,BC	2	15	#	-	-	-	0	?
ADD IX,DE	2	15	#	-	-	-	0	?
ADD IX,IX	2	15	#	-	-	-	0	?
ADD IX,SP	2	15	#	-	-	-	0	?
ADD IX,BC	2	15	#	-	-	-	0	?
ADD IX,DE	2	15	#	-	-	-	0	?
ADD IY,IY	2	15	#	-	-	-	0	?
ADD IY,SP	2	15	#	-	-	-	0	?
SBC HL,reg_pair	2	15	#	#	#	#	1	?
SBC HL,SP	2	15	#	#	#	#	1	?

Flags Notation:

indicates flag is altered by operation
0 indicates flag is set to 0
1 indicates flag is set to 1
- indicates flag is unaffected
? indicates flag is ?

Table 6. 16 bit arithmetic instructions

Chapter 9

Loops, Jumps and Block Operations

This Chapter covers two apparently different areas of the Z80 instruction set;

- (a) Instructions that cause control to pass to another part of the program, in a similar way to the GOTO and GOSUB BASIC commands pass control around a BASIC program.
- (b) The Block instructions, or Block Operators, which are operations that work on several bytes of memory simultaneously, rather than acting on single bytes.

The reason that I've grouped these instructions together is that the Block operations often involve a jump or loop operation, and so it makes sense to group the commands with the jump and loop instructions.

Jumps

All the programs that we've entered into the Amstrad so far have been able to perform their task without the presence of any machine code instructions that simulate the BASIC GOTO or GOSUB. If we were to continue this philosophy, the resultant programs would not be very powerful. The use of these machine language "GOTO" instructions thus gives us great programming power, but we must be careful when we use these instructions. Let's begin by looking at the JP, or jump, instruction which is the direct equivalent of GOTO. However, instead of jumping to a line number the JP instruction jumps to a particular address.

The JP operation has two addressing modes; immediate and Register Indirect. In the immediate mode the address is given implicitly in the command;

```
JP 40000
```

is an example of the addressing mode. In the Register Indirect addressing mode, the address is held in the HL, IX or IY register pairs. More details will be given later in the Chapter.

A jump can be UNCONDITIONAL, like the one shown above, in which case control is immediately passed to

the instructions that start at the address specified in the command. Alternatively, it can be a CONDITIONAL jump, in which case the jump is only made if some condition, indicated by the status of a flag, is satisfied. This is the machine code equivalent of

```
IF ... THEN GOTO ...
```

in BASIC. For example,

```
JP Z,40000
```

will only execute a jump to address 40000 if the result of the last operation was zero; i.e. if the Z flag is set. Other flags can also be used to decide whether to jump or not.

```
JP C,40000
```

will jump when the C flag is set. Other instructions of this type are;

```
JP NZ,address  jump if result non-zero
JP NC,address  jump if carry clear
JP P,address   jump if result positive
JP M,address   jump if result negative
JP PE,address  jump if parity even
JP PO,address  jump if parity odd
```

All these instructions are three bytes long; a one byte opcode and a two byte address. The address is stored, as we might now expect, low byte first. Thus the instruction

```
JP Z,&9C00
```

is coded as

```
CA 00 9C
```

Let's now write a simple machine code program that uses a JP instruction. However, before we start, a word of caution. Like the good soldier, when the CPU is told to jump, it jumps even if the orders given were a little silly. If we've made a mistake in specifying an address, then the jump might pass control to a byte in memory that represents a byte of data, or to the second byte of a three byte instruction, or any other place in the computer memory. The result is usually a crash. Secondly, if a condition arises that causes a sequence of instructions to be executed for ever, you often have to turn off the computer to break the loop

Example 10:

In this example, I'll list the bytes alongside the assembler instructions, so that you'll get the bytes

for the JP instruction correct.

```
LD HL,&C000      21 00 C0
LOOP LD (HL),255  36 FF
      INC HL      23
      LD A,L      7D
      OR H        B4
      JP NZ,LOOP  C2 43 9C
      RET        C9
```

This program MUST be entered into memory at address 40000, because we've specified an address in the program for the jump instruction. If we were to load the program to another location in memory without altering the address given in the JP instruction, a crash would be the most probable result. Enter the above bytes, and type CALL 40000. The screen will fill up, and the effect will be slightly different in each screen mode. Let's look at how the program works. In the Amstrad, the area of memory between address &C000 and &FFFF is reserved for screen memory. Thus writing data to this area of memory will have an obvious affect on the screen. This program loads every byte between &C000 and &FFFF with the value 255. We check for the end of screen memory by testing the HL register for the value 0, which it will assume after we increment it from a value of &FFFF. We have to test each of the separate registers that make up the HL register pair to ensure that it holds zero, because the INC HL instruction doesn't affect the Z flag.

Try to simulate this program with a FOR ... NEXT loop in BASIC, and I'm sure you will be impressed with the speed of the machine code program.

There are some occasions when machine code is too fast for a particular application. In these cases we often have to use delay loops to slow things down a little. This example shows how we can do this.

Example 11:

This program shows a machine code delay loop, which is repeated a given number of times. The inner loop is controlled by the DE register pair and the outer loop is controlled by the HL register pair. The program must once more be entered at address 40000 if you use the bytes given below, again because of the fact that we specify an address in the JP instructions.

	LD	HL,65535	21 FF FF
LOOP1	LD	DE,100	11 64 00
LOOP2	DEC	DE	1B
	LD	A,E	7B
	OR	D	B2
	JP	NZ,LOOP2	C2 46 9C
	DEC	HL	2B
	LD	A,H	7C
	OR	L	B5
	JP	NZ,LOOP1	C2 43 9C
	RET		C9

Again, note the additional instructions that are needed to check that the register pairs are zero.

You can see that the JP instructions give us the ability to produce programs whose behaviour depends upon the status of the flags. The JP instructions that we've seen so far require the programmer to specify a two byte address even if the destination of the jump is only a few instructions away from the jump instruction. There are some alternative JP instructions that only require one byte specify the destination; these are called RELATIVE JUMPS.

Relative Jumps

In the programming examples we've just seen, the destinations of the jumps were not very far from the jump instructions themselves. However, we still had to specify a two byte address. Also, the use of a specific address in a JP instruction means that the program must always be loaded to and executed at the same point in memory every time. The relative jump instructions offer us a way around these two problems. The instructions are only two bytes long; a one byte opcode and a single displacement byte which specifies where the jump is to be made to.

The displacement byte represents a number between -128 and +127. This byte represents the "distance" to be jumped by the instruction. We can thus pass control to any byte within the range 127 bytes after the relative jump and 128 bytes before it.

The mnemonic for this instruction is

JR cc,d

where d is the displacement and cc is one of the conditions that are applicable to relative jumps. We can have unconditional relative jumps of the form;

JR d

The value of the displacement byte causes a jump in the following fashion.

When the CPU encounters a JR instruction, the

first thing that occurs is that the CPU adds 2 to the current value of the Program Counter. The practical result of this is that the address referenced by the displacement byte is reckoned from the byte after the JR instructions and displacement byte. Look at the below example to make things clearer.

```

INC  A           -3
JR   Z,         -2
     02         -1

LD   A,         0
     00         +1

LD   A,         +2
     02         +3

```

The byte immediately following the displacement byte is numbered 0, the next byte 1, and so on. Similar numbering occurs in the other direction. Thus the instruction

```
JR   -2
```

is not a terribly good idea, because it causes a jump back to the JR instruction, thus causing a perpetual loop!

As you might expect, the negative numbers are stored in Two's Complement representation. Thus -2 is written as &FE. As a further example, look at the below program, which includes a label.

```

LD   A,00
LOOP INC  A
     JR  NZ,LOOP

```

The displacement byte here would be -3, or &FD. Some conditional relative jumps are possible, but not as many as for the normal JP instruction. The list below shows the conditional relative jumps that you have at your disposal.

```

JR   C,d
JR   NC,d
JR   Z,d
JR   NZ,d

```

So, if you want to make a jump based upon the parity of a number, you'll have to use a JP instruction.

The major advantage offered by the relative jump instructions over the conventional jump instructions is that they make no reference to a particular memory location; all jump destinations are specified relative to the current position of the JR instruction. This means that a program that's been written with only

relative jumps in it can run at any location in memory. Example 12 shows this in action.

Example 12:

Try the below program at addresses 40000, 41000, 42000 and see that it works at each of these locations in the same way. Don't forget to change the address in the CALL statement each time!

```

                LD   HL,&C000    21 00 C0
LOOP           LD   (HL),255    36 FF
                INC  HL          23
                LD   A,L         7D
                OR   H           B4
                JR   NZ,LOOP     20 F9
                RET              C9
```

This type of program, that will run at several different memory locations, is said to RELOCATABLE. It can run at an address in memory other than that at which it was originally written.

Register Indirect Jumps

We've already mentioned these briefly; here, we place the address to which we want the jump to be made in to either the HL, IX or IY register pair. The below command is then executed, where rr is the appropriate register pair.

```
JP   (rr)
```

Thus,

```
JP   (HL)
JP   (IX)
JP   (IY)
```

are the only legal Register indirect commands. As a concrete example, the below lines will cause the CPU to begin executing the instructions at location 0000 of the memory; this will cause a system reset.

```
LD   HL,0000
JP   (HL)
```

There are no conditional Register Indirect jumps.

FOR . . . NEXT Loops in machine code

We've now seen the machine code equivalent of GOTO and IF ... THEN statement, so let's now look at the FOR ... NEXT construction in machine code. Let's start by looking at the use of a FOR ... NEXT loop in BASIC. Generally, it is used when we wish to perform a particular set of instructions a given number of times. Look at the below BASIC program;


```

10 C=0
20 FOR I=1 TO 6
30 LET C=C+1
40 NEXT I

```

In machine code, as you might expect, we use a register to replace the I variable. The easiest way to do this with a single register is to use a count down, rather than a count up. It is easier for us to check for zero than for a non zero value. The below routine will simulate the above BASIC program.

```

                LD   C,0           initialise variable 'C'
                LD   B,6           initialise variable 'I'
LOOP           INC   C             C=C+1
                DEC   B
                JR   NZ,LOOP       simulate the NEXT command

```

The last two instructions in this short routine occur together quite often in Z80 machine code, and were united by the CPU designer to give a single command, DJNZ. The full instruction is

```
DJNZ d
```

where d is a displacement byte that is identical in form to the displacement bytes that we use with the relative jump instructions. The instruction DJNZ stands for Decrement and Jump if Non Zero. The instruction needs two bytes, and using it we can rewrite the above program as

```

                LD   C,0
                LD   B,6
LOOP           INC   C
                DJNZ LOOP

```

There is only one problem with DJNZ; it is only useful for counting loops with up to 256 passes. There is no 16 bit DJNZ command. 256 loops, you say; how can we do this when the biggest number that we can put into an 8 bit register is 255? Well, if we set the B register to zero, then execute a DJNZ instruction, the B register will be decremented, so leaving a value of 255 in the register.

DJNZ instructions can be nested to enable us to count passes through loops of instructions that need more than 256 passes. Look at the below example;

```

        LD    B,16
OUTLOOP PUSH BC
        LD    B,256
INLOOP  ...
        ...
        DJNZ INLOOP
        POP  BC
        DJNZ OUTLOOP

```

An alternative way, that uses another 16 bit register, is to use a 16 bit DEC command. We've seen this in action already. Obviously, if we wanted to include a STEP in these commands we simply add more INC or DEC commands. For example,

```

        LD    C,0
        LD    B,100
LOOP    INC  C
        INC  C
        DEC  B
        DJNZ LOOP

```

Here the C register will count up 0, 2, 4... and the B register will count down 100, 98, 96... Remember that there is one DEC B instruction hidden in the DJNZ instruction. One problem could arise with the short routine above; if you put an odd number in the B register, then you'll never actually achieve a value of zero in the B register. The looping would thus carry on permanently.

The below table features the loop and Jump instructions with their relative times. No flags are affected by these operations, with the exception of the DJNZ.

Mnemonic	Bytes	Time Taken
JP nn	3	10
JP cc,nn	3	10
JR nn	2	12
JR cc,nn	2	7/12
JP (HL)	1	4
JP (IX)	2	8
JP (IY)	2	8
DJNZ d	2	8/13

Table 7. Jump and loop instructions

Where two times are mentioned, the first time given is that time taken when the condition is NOT met, and the second time is the time taken when the condition is met. The relative jumps take a little longer than the JP instructions when a jump is actually made because the address to which the jump is to be made has to be calculated from the current address and the displacement byte.

CALL and RETURN

In BASIC, we had the instructions GOSUB and RETURN that gave us the ability to use subroutines. A subroutine, you will remember, is a block of instructions that is stored once in memory but that can be called as often as you like in a program. In Z80 machine code we have the same ability; in fact, we've already used a machine code subroutine call. Whenever we issue a CALL instruction, we are effectively making a subroutine call from the BASIC Interpreter to your machine code program. The RET with which we end our programs is the equivalent of the RETURN instruction in BASIC subroutines.

In machine code, the

CALL nn

instruction will call a subroutine at address nn. The instruction is a three byte instruction. One byte is the opcode and the others are the address of the subroutine. The bytes of the address should be entered into memory low byte first. Any piece of code that is being used as a subroutine should end with a RET instruction. Once a RET is executed by the CPU, the CPU starts executing the instruction immediately after the CALL instruction. How does the CPU know where to return to?

Well, the stack is used. This is the main role of the stack in the Amstrad computer. When the CALL is made, the CPU saves the address of the first instruction after the CALL instruction on to the stack. The RET instruction, when executed, looks at the last entry on the stack and effectively POPS it in to the program counter. The RET instruction will thus cause the CPU to jump back to the address that is represented by the last entry on the stack.

Within the body of the subroutine, therefore, it is vital that all PUSHes on to the stack are balanced by POPs from the stack, if the RET instruction is to return the CPU to the instruction following the CALL. There are some techniques in which the item to be treated as the return address can be altered, but these are best left alone until you've gained some experience. We'll look at one of these techniques shortly. The normal behaviour of CALL and RET is shown below.

```
40000 CALL 41000 → 41000 INC A
40003 INC A ← 41001 RET
```

When we include a PUSH and POP in the subroutine, we have the below situation.

```

40000  CALL 41000  → 41000  PUSH AF
        INC A      ←          INC A
                                POP AF
                                RET

```

The PUSH is balanced by a POP, and so the RET instruction receives the correct address off of the stack.

Below, however, we have an "unbalanced" PUSH operation, which leaves the stack altered from what the FET expects to find.

```

40000  CALL 41000  → 41000  LD BC,0000
        INC A      ←          PUSH BC
00000  ←          RET

```

The CPU will now start executing the instructions at address 0 as soon as the RET is executed. This will reset the computer. However, in some applications, changing the address that the RET instruction will find on the stack can be useful, though it is certainly NOT good programming practice.

This type of behaviour effectively uses the RET instruction as a kind of jump, and the below routine shows this in action;

```

40000  CALL 41000  → 41000  POP DE
                                LD BC,42000
                                PUSH BC
42000  ... ←          RET

```

The POP DE instruction removes the original return address from the stack. We then PUSH the desired return address onto the stack, and the RET instruction causes the jump to be made. All this is very interesting, but is not the real use of the CALL instruction, which is to call a subroutine and return to the instruction following the subroutine call! You will note that the full address is required in the CALL statement; there are no relative subroutine calls. Programs written with subroutines in them are not very relocatable, unless they only use Amstrad ROM routines in the subroutine call.

Saving Registers

You will occasionally want to make a subroutine call but still preserve the contents of certain register pairs; we can do this by PUSHing the register pairs onto the stack before we make the CALL, and POPing the registers back after the CPU has executed the subroutine. Saving the registers in this way is called PRESERVING the CPU registers, or, if you're into impressive sounding phrases, "saving the CPU environment".

Let's now examine a subroutine call in a short program.

Example 13:

This routine simply uses an Amstrad ROM routine that waits for a key to be pressed before proceeding with the program. Load the program to address 40000, and the bytes given below will be correct.

```
          CALL SUBR          CD 44 9C
          RET                C9
SUBR     CALL &BB18         CD 18 BB
          RET                C9
```

There are effectively two subroutine calls here, one to the label SUBR and one to address &BB18. CALL 40000 will cause the machine to pause until you press a key.

A few general points about subroutines. Always try and put the definitions in such a place in memory that they won't be accidentally executed by the CPU without them being CALLED. Subroutine calls are slower than having the code repeated wherever it is needed in the program, as the CALL and RET instructions take a finite time to execute. Subroutines, although they save memory, take more time. In any application where time is important but memory isn't, I tend to use repeated chunks of code throughout the program.

Conditional Subroutine Calls

In BASIC we use

```
IF ... THEN GOSUB nn
```

when we wish to conditionally call a subroutine. A similar structure exists in machine code.

```
CALL cc,address
```

will execute a CALL to a given address only if a particular condition is satisfied. The conditions that can be used are as follows:

```
CALL C,address  call if carry set
CALL NC,address call if carry clear
CALL Z,address  call if result zero
CALL NZ,address call if result not-zero
CALL PE,address call if parity even
CALL PO,address call if parity odd
CALL M,address  call if result negative
CALL P,address  call if result positive
```

No flags are affected by the CALL instruction. We can thus simulate the ON ... GOSUB statements of BASIC by:

```
LD  A,(CHOICE)
CP  1
CALL Z,OPTION1
CP  2
CALL Z,OPTION2
```

and so on. One thing to note is that on entering the option selected by the appropriate CALL command, the A register should be preserved. If this isn't done, on return from the option the CPU could possibly enter another option after returning from one.

We can also have conditional RET instructions, that will only cause a return from the subroutine when a particular condition is met; the conditions catered for by this instruction are the same as for the conditional CALL statement.

Mnemonic	Bytes	Time Taken	Effect on flags					
			C	Z	P/V	S	N	H
CALL address	3	17	-	-	-	-	-	-
CALL cc,address	3	10/17	-	-	-	-	-	-
RET	1	10	-	-	-	-	-	-
RET cc	1	5/11	-	-	-	-	-	-

Table 8. Call and return instructions

Where two times are shown in the above Table, the shorter of the two times is that taken when the condition is not met.

Restarts

These can be seen as 1 byte subroutine calls; the catch is that they can only call addresses within the first 256 bytes of the 280's memory map. They are thus in that area of memory that is used by the Amstrad Operating System, and so we are denied access to them for our own programs.

The role of the restart, or RST commands, is to provide fast access to a few routines that will be commonly used in a program. This is why the Operating System has first choice. Of course, we could use the calls, but there is not space in this book for a complete explanation of what the Amstrad uses the various RST instructions for.

There are 8 RST instructions, which enable us to call 8 separate addresses. The addresses to which we have access are &00, &08, &10, &18, &20, &28, &30, &38. For example the command

```
RST &00
```

will cause a jump to address 0, which is the reset address for the Amstrad computer system. The restart at address &30 is reserved for the user to program,

but it's best left alone until you've gained some experience.

Interrupts

These are particularly useful things to have in computers; an interrupt is a signal sent to the CPU to instruct it that some situation has arisen in the computer that requires the immediate attention of the CPU. The CPU makes a note of where it is in it's work, and then jumps off to a routine that deals with the situation. This routine is called an Interrupt Service Routine, or ISR. It will usually save various registers, perform the task, restore the registers and return the CPU to the task that it was previously doing. The interesting thing is that the user isn't usually aware that anything has happened! To return from an Interrupt Service Routine, a special command is used. This is

RETI

and is a special form of RET. The commonest form of interrupt on the Amstrad is the one that is used to call the various Operating System routines to read the keyboard. This is called 50 times a second, and executes a RST &38 instruction.

Interrupts of the type that we've mentioned so far have been what are called MASKABLE INTERRUPTS; this means that we can instruct the CPU to ignore them. A second class of interrupts, called NON MASKABLE INTERRUPTS cannot be ignored by the CPU. The command

DI

causes the CPU to ignore all maskable interrupts. The command

EI

makes the CPU start taking notice of the interrupts again. These should not be mistaken with the DI and EI commands that are available from BASIC; these deal with other things. EI stands for Enable Interrupts, and DI stands for Disable Interrupts. You probably won't be experienced enough to fiddle around with interrupts on the Amstrad, but if you should disable interrupts within your program, it's vital to re-enable them with EI before executing the final RET instruction.

That completes this review of instructions that pass control around the program. We'll now go on to look at a rather powerful range of instructions that use jumps to operate on more than one byte automatically. They are called the BLOCK INSTRUCTIONS.

Block Operations

These instructions operate on several bytes rather than just the usual one or two bytes. However, the simplest of the block instructions do only work on single bytes.

The CPI Instruction

The simplest is the

CPI

instruction. This stands for ComPare and Increment. This instruction compares the contents of the A register with the contents of the byte addressed by the HL register. The HL register pair is then automatically incremented. It thus performs the

```
CP (HL)
INC HL
```

instructions. The obvious use for this command is in searching through the memory of the computer for a given byte. Example 14 shows this in action.

Example 14:

The assembler instructions are as follows:

```
LD E,(IX+0)
LD D,(IX+1)
LD A,(DE)
LD HL,1000
SEARCH CPI
JR NZ,SEARCH
DEC HL
LD A,L
LD (DE),A
LD A,H
INC DE
LD (DE),A
RET
```

Type in the bytes at address 40000. The displacement for the JR instruction is -4, or &FC. This is because the CPI instruction is a two byte instruction. The program searches through memory from location 1000 onwards. It can be called with

```
CALL 40000,@A%
```

where A% holds the byte that you're looking for. On return, A% will hold the address at which the first occurrence of that byte was found. The DEC HL instruction after the JR NZ instruction is needed because of the automatic incrementing of the HL register. This doesn't affect the flags, and so if the

Z flag is set it must be due to the CP instruction finding a match.

The CPD Instruction

A similar instruction, CPD, performs a similar job but decrements the HL register pair instead of incrementing it. As well as modifying the contents of the HL register pair, both CPI and CPD decrement the contents of the BC register. This quite usefully allows us to search through a block of bytes of a given length, such as a data area of a program. Example 15 shows this in operation.

Example 15:

Let's look at the program first, then describe it. Enter the code to address 40000, then call it with

```
CALL 40000,@A%
```

where A% holds the appropriate value. The program searches 255 bytes starting from address 1000 in memory.

```
LD BC,255
LD E,(IX+0)
LD D,(IX+1)
LD A,(DE)
LD HL,1000
SEARCH CPD
JR Z,OUT
INC C
DEC C
JR NZ,SEARCH
OUT INC HL
LD (DE),A
LD A,L
INC DE
LD A,H
LD (DE),A
RET
```

Because the 16 bit DEC operations, which are implicit in these instructions, don't bother the flags, we have to test the BC register contents ourselves to see whether the register pair contains zero or not. As we're only counting 255 bytes here, we use the INC C and DEC C instructions to see if the C register is holding zero. The DEC instruction will set the Z flag if the C register was originally zero before the INC C operation. The program returns the value 255 if the byte searched for is not found, or the address of the byte if it is found.

CPIR and CPDR

These two instructions are really powerful; they are two byte instructions and are the equivalent of a CPI or CPD with a built in jump instruction!

The CPU automatically searches a block of memory until either a match is found or the end of the block is found. The A register specifies the byte to be searched for, the HL register holds the start address of the block to be searched and the BC register pair holds the number of bytes to be searched. The instruction will terminate for one of two reasons;

- (a) A match has been found.
- (b) The block end has been reached.

Thus after a CPIR or CPDR instruction we must test to see which of these conditions has caused the termination of the instruction. This isn't such a difficult task as it sounds; simply remember that if the block has been totally searched the BC register will contain the value 0. It's thus simply a matter of testing for this fact. The below piece of code shows this in action.

```
LD    HL,42000
LD    BC,1000
LD    A,255
CPIR
LD    A,B
OR    C
JR    Z,END_FOUND
```

The label END_FOUND would be jumped to if the BC register was equal to zero on termination. Otherwise, the termination of the CPIR command is due to the finding of a match.

However, these instructions are fairly time consuming, but they are still faster than doing each of the "bundled" operations individually.

Block Moves

Occasionally we may want to move whole chunks of memory around. One way to do this would be to use a piece of machine code like the one listed below;

```
LD    HL,40000
LD    B,200
LD    DE,42000
LOOP  LD    A,(HL)
      LD    (DE),A
      INC  HL
      INC  DE
      DJNZ LOOP
```

Here we transfer 200 bytes from address 40000 to

42000. This is effectively a copy operation; the bytes will still be present at address 40000 onwards. HL register pair points to the byte that we're copying, and DE points to where in memory that byte is to be written to. The B register is used to count the number of bytes that we want to transfer. This program is totally workable, but inefficient, as there is an instruction in the Z80 instruction set to do this kind of operation for us automatically. The first instruction of this type that we'll look at is called LDI.

The above routine can be rewritten as

```

                LD    HL,40000
                LD    DE,42000
                LD    BC,200
LOOP           LDI
                LD    A,B
                OR    C
                JR    NZ,LOOP

```

The DE register pair is often called the DESTINATION register and the HL register pair the SOURCE register. Once LDI is executed, the HL and DE registers will be incremented and the BC register will be decremented. The command LDD does a similar job, but here the HL and DE register pairs are decremented instead of incremented. The above routine can be implemented in a more efficient manner if we know that after an LDI or LDD instruction, the P/V flag is set if the BC register pair DOES NOT contain zero. We can thus use this flag to see if a repeat is needed or not.

However, there is a much more efficient method of getting an LDI or LDD instruction repeated; this is to use either the LDIR or LDDR instruction, which test the P/V flag automatically and jump accordingly.

For example, the short program below will transfer 1000 bytes from address 40000 to address 42000.

```

LD    HL,40000
LD    DE,42000
LD    BC,1000
LDIR
RET

```

The below table shows the block instructions and their effects on flags and timings.

Mnemonic	Bytes	Time Taken	Effect on Flags					
			C	Z	P/V	S	N	H
LDI	2	16	-	-	#	-	0	0
LDD	2	16	-	-	#	-	0	0
LDIR	2	21/16	-	-	0	-	0	0
LDDR	2	21/16	-	-	0	-	0	0
CPI	2	16	-	#	#	#	1	#
CPD	2	16	-	#	#	#	1	#
CPIR	2	21/16	-	#	#	#	1	#
CPDR	2	21/16	-	#	#	#	1	#

Flag Notation:

indicates flag is altered by operation
0 indicates flag is set to 0
1 indicates flag is set to 1
- indicates flag is unaffected

Table 9. Block search and move instructions

Timing:

For repeat instructions, the times shown are for each cycle. The shorter time indicated is for the case of the instruction terminating - e.g. for CPIR, either BC = 0 or A = (HL).

Chapter 10

Ins and Outs and Odds and Ends

In this Chapter we'll take a very brief look at the Z80 Input and Output instructions, and we'll also examine a few final instructions that don't fit into any firm category.

Input and Output Instructions

As well as being able to communicate with the RAM and ROM, the CPU can also read information from and write information to a variety of addresses called INPUT/OUTPUT or I/O addresses. There are 65536 of these available to the CPU, and they are used by the CPU to enable it to communicate with the many other electronic devices that make up the Amstrad Computer, such as the Gate Array or the PSG.

The actual way in which this is done is beyond the scope of this book, and the instructions mentioned below should only be used if you have a sound knowledge of the Amstrad I/O system. IT IS POSSIBLE TO DAMAGE the system if you mess around too much, although a more likely result is a system crash!

The instructions used by the Z80 to communicate with these I/O devices are called Input and Output instructions, and there are several of them available to the programmer. However, due to the way in which the Amstrad Hardware is arranged, there are only a couple of instructions that can be used with absolute safety - the others often crash the system. Anyone attempting to read or write from I/O locations should be aware of the problems involved, and never forget that damage to the system is possible. A good I/O description of the Amstrad can be found in Don Thomasson's book "The Ins and Outs of the Amstrad", also published by Melbourne House.

The I/O instructions that are usable with the Amstrad hardware are as follows:

```
IN      r,(C)
OUT     (C),r
```

where r is an 8 bit register. The IN instruction reads a byte of data from the I/O device to the CPU

register; it is similar to loading a register from memory. The OUT instruction sends a byte from the CPU register to the I/O location. This is the same sort of operation as loading a memory location from a CPU register. The address of the I/O location of interest here is held in the BC register pair. Care is needed here, and an I/O map is essential, as not all addresses are used by the System. An example of the use of the IN instruction is shown below:

```
LD      BC,&BE00
IN      A,(C)
```

The data will be read into the A register from, in this case, the CRTIC. An IN instruction involving the A register also affects the S, Z and P/V flags. Again, you can crash the system if you read from certain I/O addresses.

Just as we have the LDIR instructions for memory transfers, we can have block transfer operations for the IN and OUT operations. However, due to the arrangement of the Amstrad Hardware, they CANNOT be used on the Amstrad.

On the whole, these instructions are of minimal use on the Amstrad; apart from carrying the risk of possible damage, the Amstrad ROM routines offer us all we are usually likely to need in terms of communicating with the peripheral devices of the System.

Odds and Ends

I want to use the rest of this Chapter to mention a few instructions that are either rarely used or don't fit in anywhere else! Hence the "odds and ends" part of the Chapter title.

The first instruction that I want to look at is really useful, despite the fact that it does nothing!

NOP

The NOP instruction, when encountered by the CPU, just causes the CPU to "mark time" for a while. This rather pointless sounding activity can be rather useful if we want to slow things down a little. The NOP instruction could easily be included in the time delay loops that we saw in a previous Chapter.

A second use is to delete instructions from a program by replacing them with the opcode for NOP, which is 00. Why do this? Well, if the program contains any jumps, deleting instructions usually results in the relative jump displacements or the absolute jump addresses being incorrect. If we simply replace each byte of the offending instructions with

"00" then the jumps will be correct because the number of bytes in the program will not have changed.

You can see that, for an instruction that does nothing, it's surprisingly useful!

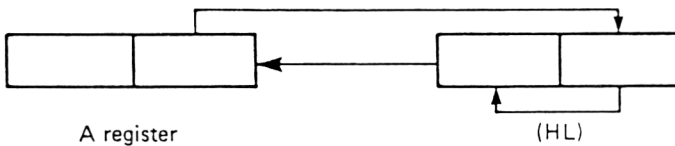
The next two "odd" instructions that we'll look at here are called RRD and RLD. They aren't commonly used instructions, but are useful if you're dealing with BCD numbers.

RRD and RLD

We saw some time ago how we could use shift instructions to affect the value held in registers. We also saw the Rotate instructions. These instructions all worked on single bits within a byte. These two instructions work on nibbles within a byte! The instructions only work in the Register Indirect Addressing mode. In both instructions, the HL register pair contains the address of the byte in memory that is to be manipulated. The operations of the two instructions are shown below. Note how the A register is also used.

Operation of RLD

In general terms this is



For a particular example, let's examine the below situation.

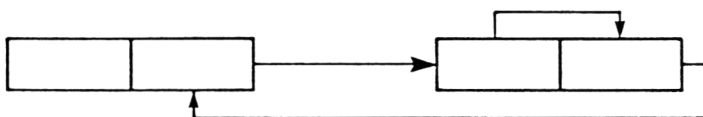
A=0010 1100 (HL)=1010 0010

After the RLD instruction, we're left with

A=0010 1010 (HL)=0010 1100

Operation of RRD

In general terms, this is



For a particular example, we can consider the below situation.

A=1010 0001 (HL)=0010 0110

After the RRD command, the bytes are

A=1010 0110 (HL)=0001 0010

As I said, these commands are really only useful if you're doing some complicated manipulations of BCD numbers.

HALT

An instruction whose name is very descriptive: this simply causes the CPU to stop what it's doing, or HALT, until it receives an interrupt. As there's an interrupt at least once every 300th of a second on the Amstrad, the CPU won't halt for long!

Interrupt Modes

The Z80 can respond to Interrupts of the maskable kind in a variety of ways. The ways in which the CPU responds are called Interrupt Modes, and there are 3 of these. They are called IM0, IM1 and IM2. The Amstrad runs in Interrupt Mode 1, which causes the CPU to execute an Interrupt handler routine at address &38 whenever an interrupt occurs.

The instruction

IM 0

will change the Interrupt mode to Mode 0. IM 1 and IM 2 will change the interrupt mode in use to the corresponding modes. There is not really room in this book to go into details about these different modes, but it's best not to alter them until you know exactly what you're doing!

NEG

This instruction carries out an automatic complement and increment operation on the A register, thus

negating the A register contents. Thus, the instructions

```
LD    A,3  
NEG
```

will leave the two's complement representation of -3 in the A register. The flags are affected in the following fashion. C = 0 if the original value was zero. If the original value was 128, then C = 0 AND the P/V flag will be set. Otherwise C is set to zero. The Z and S flags assume a value depending upon the result of the operation.

The I and R registers

These are two special purpose 8 bit registers. The I register is concerned with interrupts, and is nothing to do with the IX and IY registers. The R register is called the REFRESH register, and both these registers are of use only to the experienced programmer.

We've now looked at all the Z80 instructions. We can now go on to see how we can use the built in facilities of the Amstrad computer, starting with a look at the Amstrad sound chip.

Chapter 11

Amstrad Sound

All the sounds on the Amstrad are generated by a device called the AY-3-8912 Programmable Sound Generator. These devices are often used in home computers to generate sounds; the CPU simply instructs the Sound chip to produce a particular sound, and then it can go off and do something else while the PSG continues to generate the sound independently. In older systems that do not use these sound chips, the CPU itself has to produce the sounds and is thus not able to do anything else in this period.

Although the Sound chip provides the "raw" sounds for the Amstrad, many of the special effects that are possible are controlled by the CPU running programs. In this section, I will look at the Sound Chip in terms of how it can be programmed to provide some basic sounds, and give you enough information to write your own sound effects programs.

General Notes on the AY-3-8912

This chip is called a 3 channel device; that is, it is capable of playing three tones simultaneously, each tone being of a different pitch and amplitude to the others. Amplitude is just the technical term for volume of sound. Each channel can also play "noise", and so the device is well suited for producing sound effects. Like the CPU, the PSG contains registers, which are used to control the nature of the sound being produced. Of course, we can't use these registers to do arithmetic in! As well as these control registers, there is also an input/output register, which enables the PSG to communicate with other electronic devices in the computer system. In the Amstrad, this register is used, under CPU control, to get information from the keyboard. Under normal conditions, the PSG is controlled by the programmer accessing 3 locations in the I/O map of the computer. However, in a machine with the complexity of the Amstrad, it's safer to use a machine code routine contained in the Amstrad ROM to communicate with the PSG, and thus control it. The reason for this is that because the keyboard is read 50 times a second, the PSG might be accessed by the OS half way through one

of your operations, and this could really confuse things! Using the provided ROM routine, which is called MC SOUND REGISTER, will ensure that nothing unpleasant happens!

PSG Registers

There are 15 registers in the PSG; 14 of these are concerned with sound generation and the other one is the I/O register that we've just mentioned. Let's now go on to look at how we use the ROM routine to access one of these registers. They are numbered from 0 to 14, and register 14 is the I/O register.

Writing to PSG Registers

This is simplicity itself, thanks to the ROM routine. We simply load the register number of interest into the A register of the CPU and the value that we want to write to that register into the C register of the CPU. Then we simply make a CALL to the ROM routine, which is accessed at address &BD34.

As an example, let's say that we want to send the value 4 to PSG register 8. We simply use the below code;

```
LD   A,B
LD   C,4
CALL &BD34
```

Easy isn't it?

Example 16:

This short routine enables us to write values to PSG registers from BASIC.

```
LD   C,(IX+0)
LD   A,(IX+2)
CALL &BD34
RET
```

If the code is entered at address 40000, then

```
CALL 40000,2,3
```

will write the value 3 to PSG register 2. As we don't need to return any values to BASIC from this routine, we don't have to use the "@" symbol.

Of course, the crunch is knowing what to write to each register to get the desired effect. So, let's get down to finding out what each register does.

The only problem with the PSG is that several registers are involved with the production of a sound. We'll look at those involved with tone generation

first, and then examine those to do with noise generation and sound modulation.

Registers 0 and 1

These are treated together by the PSG, and together they hold a 12 bit number which represents the pitch of the tone played on Channel 1 of the PSG. The lower 8 bits of the pitch value are held in Register 0 and the upper 4 bits are held in the lower 4 bits of Register 1. The upper 4 bits of register 1 are not used. It's thus clear that the contents of the lower 4 bits of register 1 has a higher significance to the value of the pitch than does the contents of register 0. For this reason, Register 1 is called the COARSE TUNE CONTROL REGISTER, and Register 0 is called the FINE TUNE CONTROL REGISTER. The higher the overall value held in the twelve bit register is, the lower pitched the tone generated on Channel 1 is. So, to write a value to these two registers, we'd use code like that shown below.

```
LD  A,0
LD  C,data for R.0
CALL &BD34
INC A
LD  C,data for R.1
CALL &BD34
RET
```

Registers 2 and 3

These are also pitch control registers, but they control the pitch of the tone played on Channel 2 of the PSG. They work in a similar fashion to Registers 0 and 1. Here, Register 2 is the Fine Control Registers and Register 3 is the Coarse Control Register.

Registers 4 and 5

These registers are the Pitch control registers for channel 3 of the PSG. Register 4 is the Fine Control Register for this Channel and Register 5 is the Coarse Control Register.

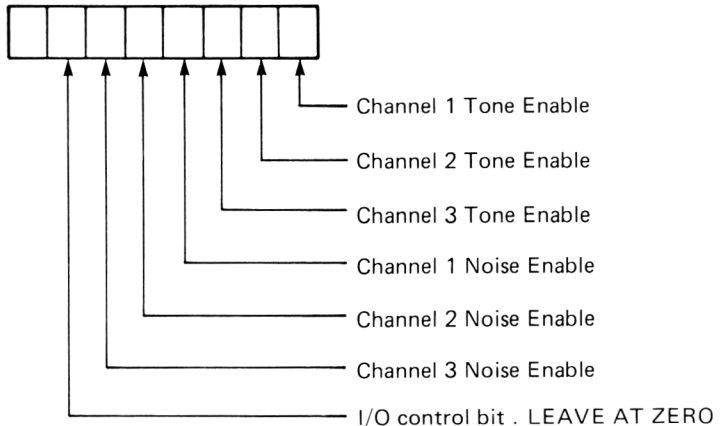
Register 6

Register 6 is not concerned with tone generation, and so it will be examined later in the Chapter.

Register 7

This is best looked at as the main control register of the PSG. Unless various bits of this register are set in the correct fashion, no sounds will be produced by the PSG, no matter what values are placed in the other PSG registers, so it's obviously quite important. Each

bit, except one, controls one aspect of PSG behaviour. Bit 7 is not used by the PSG. The aspects of behaviour controlled by different bits of the register are shown below.



Let's now examine these bits in more detail.

Bit 0

This is called the Channel 1 tone enable bit. When this bit is set at zero, a tone can be played on Channel 1 provided that it has a volume that is loud enough to be heard! If set to 1, however, no tone can be played. When set to 1, Channel 1 is said to be DISABLED, and when this bit is set to zero the tone on Channel 1 is said to be ENABLED. It effectively turns the tone on and off.

Bit 1

This is this Enable/Disable bit for Channel 2 and is similar in function to Bit 0.

Bit 2

This is the Enable/Disable bit for Channel 3, and is similar in function to bit 0.

Bit 3

This is the Channel 1 Noise Enable Bit, and it's status turns any noise on Channel 1 on or off. The subject of noise will be considered in detail shortly. When set to 0, noise will be generated on Channel 1 at a volume that depends upon the amplitude that we've set for that channel. When set to 1, no noise will be

generated. It is possible to have both bits 0 and 3 to 0. Setting both these bits to 1 will completely inhibit all sound output on Channel 1.

Bit 4

The Channel 2 noise Enable/Disable bit. It is similar to Bit 3 in function.

Bit 5

The Channel 3 noise Enable/Disable bit. It is similar in function to Bit 3.

Bit 6

This controls whether the I/O register is set to be an Input Register or an Output Register. This bit should be left set to zero, to signify that the I/O register is to be used as an Input Register for the keyboard. Setting this bit to 1 will disable the keyboard, and the only way to recover from this, unless your program resets the bit to zero before finishing, is to turn the power off!

Bit 7

This isn't used in this particular PSG.

Because of the importance of Bit 6, it's advisable to be careful when writing to this register. Always leave the top two bits, bits 6 and 7, set to zero. The other bits should be set depending upon what you want to do with the PSG.

Amplitude Control

The amplitude, or volume, of a sound determines its loudness. There are two ways of controlling the volume of sound produced by the Amstrad. The most obvious is to use the volume control on the side of the keyboard! However, this isn't exactly programmable, so we must use the three AMPLITUDE CONTROL REGISTERS that the PSG has. There is one such register for each Channel on the PSG and they are all 4 bit registers. This gives us 16 different levels of loudness, from 0, which is silence, to 15, which is the loudest volume. The Amplitude Control Registers are Registers 8 to 10. Once we've discussed these registers, we're in a position to actually make some sounds using the PSG.

Register 8

This is the Amplitude Control Register for Channel 1. We've just said that this is a 4 bit register; well,

that's not strictly true. There is a fifth bit, but that's not concerned with setting the amplitude at a constant level.

Register 9

This is the Channel 2 amplitude control register.

Register 10

This is the Channel 3 amplitude Control register.

Let's now look at the actual business of generating a tone using the PSG. There are three main steps to doing this. These are;

- (a) Set the Pitch up for that Channel.
- (b) Set the Amplitude up for that Channel.
- (c) Enable Tone on that Channel.

Example 17 shows a program for generating a tone on Channel 1.

Example 17:

Enter the code at address 40000, and run it with CALL 40000.

```
LD   A,0           select register 0
LD   C,&12         value for pitch registers
CALL &BD34
INC  A             select register 1
CALL &BD34
LD   A,8           select register 8
LD   C,15         full volume
CALL &BD34
LD   A,7           select register 7
LD   C,62         only enable tone on channel 1
CALL &BD34
RET
```

Calling this routine will probably induce you to turn the volume on the computer down a little! You will also notice one thing; the sound doesn't stop! We must deliberately do this by disabling the Tone on Channel 1, or setting the contents of Register 8 to zero. The difference between these two is that while tone is disabled, the noise can carry on if it is enabled at the same time as tone. Example 18 shows how we can use a machine code delay routine in a program that generates a tone for a given length of time before disabling it again.

Before we look at this, I'd better tell you how to stop the tone produced a moment ago! A quick press of the "CLR" button when the "Ready" prompt returns usually does the trick.

Example 18:

Enter the code to address 40000 and call it at the same address.

```
LD    A,0
LD    C,&12
CALL  &BD34
INC   A
CALL  &BD34
LD    A,8
LD    C,15
CALL  &BD34
LD    A,7
LD    C,62
CALL  &BD34
LD    B,4           set up for a delay loop
DELAY1 LD  HL,&FFFF
DELAY2 DEC  HL
LD    A,H
OR    L
JR    NZ,DELAY2
DJNZ  DELAY1       tone period
LD    A,7
LD    C,63         disable tone
CALL  &BD34
RET
```

The delay loop that is used here is the same as that which we saw in a previous Chapter. Example 19 shows how we might generate a "fade out" of the sound. It does this by repeatedly reducing the value held in the Amplitude Control register for Channel 1.

Example 19:

```
LD    A,0
LD    C,&12
CALL  &BD34
INC   A
CALL  &BD34
LD    A,8
LD    C,15
CALL  &BD34
LD    A,7
LD    C,62
CALL  &BD34
LD    C,15           initialise for fade
LD    B,15           number of steps in fade
LOOP2 LD    HL,&FFFF    tone step duration
LOOP1 DEC   HL
LD    A,H
OR    L
JR    NZ,LOOP1      tone played at volume 'C'
DEC   C
LD    A,8           decrease volume
PUSH  BC           send to PSG register 8
CALL  &BD34        push BC because this call
POP   BC           messes up the registers.
DJNZ  LOOP2        repeat with new volume
RET                                all done
```

It's also possible to vary the contents of the Tuning registers for a given note while a note is being played on that Channel. This is how the Amstrad can generate it's "Tone Envelopes" in BASIC. Example 20 shows this in action.

Example 20:

This routine generates a series of tones by simply modifying the contents of the Coarse Tune register for the Channel on which the sound is being played; thus in this example we modify the Coarse Tune Register for Channel 1, which is Register 1.

```

LD    A,0
LD    C,15
CALL  &BD34
INC   A
CALL  &BD34
LD    A,8
LD    C,8
CALL  &BD34
LD    C,15      initialise for tones
LD    B,15      number of tones
LOOP2 LD    HL,&FFFF duration of each tone
LOOP1 DEC   HL
LD    A,H
OR    L
JR    NZ,LOOP1
DEC   C
LD    A,1
PUSH  BC
CALL  &BD34
POP   BC
DJNZ  LOOP2
LD    A,7
LD    C,63
CALL  &BD34      turn off tone
RET

```

Note that this time we have to turn off the tone. You might like to try writing a machine code program that does a similar trick with the Fine Tune registers, so that you can get a smooth change from one tone to another.

So far, we've used Channel 1 all the time. The information given so far is equally applicable to the other channels, provided that you use the correct registers for that Channel. As was mentioned at the start of the Chapter, you can play tones on three channels simultaneously if you want to, by setting the various registers up and then enabling tone on whatever channels you want to use.

Noise

We briefly mentioned noise earlier in the Chapter. Let's look at it in greater detail. Noise, or WHITE NOISE as it is some times called, is best described in non-technical terms as a rushing, hissing noise, similar to that that can be heard on a VHF radio receiver when no stations are being received. Many natural, and man-made, noises have a high proportion of this type of sound in their make up. Examples are rain, wind and explosions. Noise can be played on any of the three channels, either at the same time as the tone or instead of the tone on that channel. The amplitude of noise on a particular channel is controlled by the Amplitude Control Register for that channel. The practical result of this is that you can't have noise played at a different volume to tone

on a given channel. Before you can actually hear noise on a channel, the relevant bit in Register 7 must be set to zero. Thus to enable the playing of noise on Channel 1 we must set bit 3 of this register to 0. Example 21 plays noise on Channel 1. To stop it, press "CLR" a couple of times.

Example 21:

Enter the code to address 40000 then call it with CALL 40000.

```
LD   A,8
LD   C,8
CALL &BD34      set the amplitude up
LD   A,7
LD   C,&37
CALL &BD34      enable Channel 1 noise only
RET
```

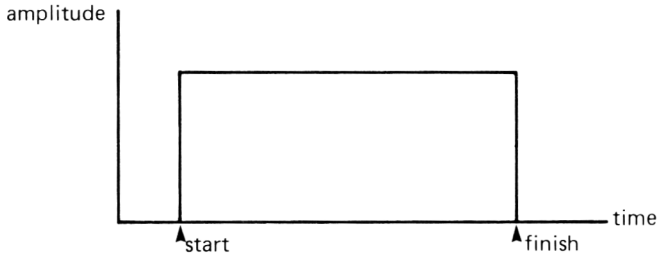
Just as a tone has a pitch, so does noise. The pitch of white noise is measured in terms of the relevant amounts of high and low frequency noise present in the sound. High frequency noise is very "hissy", and low frequency noise is more of a "rushing" sound. The pitch of the noise played by the PSG is controlled by PSG register 6, which is the Noise Pitch Control Register.

Register 6.

The fact that there is only one register for the control of the pitch of the noise generated indicates that all channels will play noise at the same pitch. The register is a 5 bit register, thus giving pitch values of between 0 and 31. A value of 0 gives the highest pitched noise and 31 gives the lowest pitched noise. Again, if we alter the value held in this register while the white noise is being played, you will hear the pitch of the noise alter.

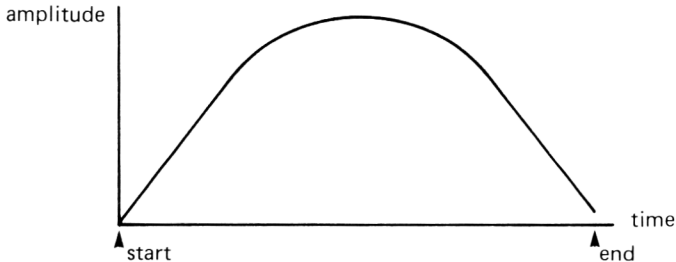
Envelopes

Nothing to do with sending letters through the post. What is the difference between the note of 'C' played on a flute and the note of 'C' played on a piano? Although the pitch is the same, the notes sound totally different. Well, as well as pitch the sound of a note depends upon its amplitude and upon the way in which the pitch and amplitude of a note change as the note is played. For example, we can represent the way in which the amplitude of a note changes with time using a graph as shown below.



This is the type of note that we've played so far.

The volume of the tone goes from zero to the maximum amplitude set as soon as the note starts, stays at this volume until the note finishes and then drops back to zero. If we could somehow "shape" the sound amplitude, we'd be able to produce more interesting sounds. Examine the below graph.



This gives us a gradual increase in amplitude, followed by a slow decrease in the amplitude of the sound. This sound shape is called an AMPLITUDE ENVELOPE. We could, of course, generate such an amplitude envelope by varying the contents of the Amplitude Register, but this would require the full attention of the CPU. Fortunately, the PSG has the ability to provide a few specific envelopes automatically. It is also possible to provide TONE ENVELOPES by varying the contents of the Pitch Tune registers for a particular channel, but this would require CPU attention whenever the tone needed changing. This is how the Amstrad provides its tone envelopes from BASIC. However, we're going to concentrate on the envelopes built in to the PSG in this Chapter. The essential thing to remember about these envelopes provided by the PSG is that once we've signaled to the PSG that we want to use one of the 8 available "hardware" envelopes, and told the PSG which one we want to use, the PSG will get on with it without any further interference from us.

The type of envelope applied to the tones played on a channel, or the noise for that matter, depends upon the contents of PSG register 13.

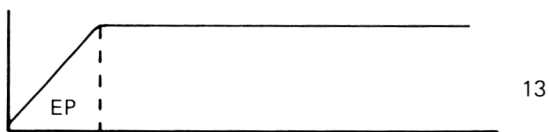
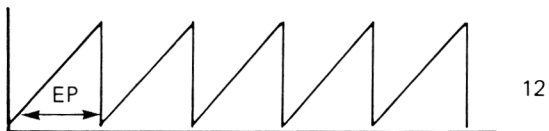
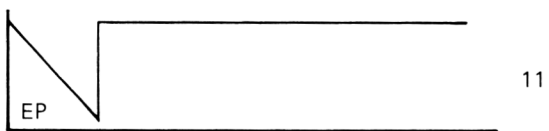
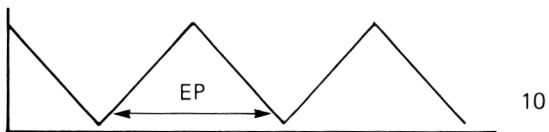
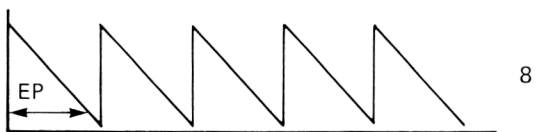
Register 13

This is the PSG Envelope Shape Control Register, and only the lower 4 bits of this register. You might think that this gives us access to 16 different envelopes; well, it doesn't. Only 8 different hardware envelopes are available on the Amstrad, but this is a limitation of the PSG rather than the computer. Writing a value to this register other than those listed below will result in one of the listed envelopes being applied to the sound. One thing that is important is that the existence of only one Envelope Shape Control Register for all three channels obviously results in the same envelope being applied to sounds on any channel that are played under envelope control.

We can see how to set up what envelope we want to use, but how do we tell the PSG that we want to play a sound on a particular channel under envelope control rather than at the fixed amplitude set by the Amplitude Control Register for that channel?

Well, remember how I mentioned that the Amplitude Control Registers had a fifth bit? This is the role of the fifth bit. It signals to the PSG whether the channel plays it's sounds under envelope control or under the control of the Amplitude Control Register. If bit 4 of one of the amplitude Control Registers is set to 1 then sound on that channel is played under envelope control. If it is set to 0, then the sound will be played at the volume specified by the lower 4 bits of the Amplitude Control Register. As a concrete example, to instruct the PSG to play it's channel 1 sounds under envelope control, we set register 8 to hold the value 16.

The envelope shapes given for a particular value of Register 13 are shown below.



In all these examples, EP stands for ENVELOPE PERIOD, and is a measure of the time it takes to execute that part of the envelope. We vary this parameter, and hence vary the rate of change of the amplitude, by altering PSG registers 11 and 12.

Registers 11 and 12

Together these form a 16 bit register pair within the PSG. Register 11 holds the 8 low bits of the value and Register 12 holds the 8 high bits of the value. The higher the value is held in these two registers, the longer is the envelope period.

We can now see a demonstration (or should it be hear) of the hardware envelope facilities of the PSG. Example 22 plays a tone on channel 1 under the control of Envelope number 14.

Example 22:

This routine will play the enveloped tone until "CLR" is pressed. Load the bytes to address 40000 and run the program with CALL 40000. You will be able to hear each amplitude change due to the long Envelope Period set by the contents of Registers 11 and 12.

```
LD    C,16
LD    A,8
CALL  &BD34      set up channel 1 for envelope
LD    A,13
LD    C,14
CALL  &BD34      set up envelope number 14
LD    A,11
LD    C,255
CALL  &BD34      set up low byte of EP
LD    A,12
LD    C,255
CALL  &BD34      set up high byte of EP
LD    A,7
LD    C,62
CALL  &BD34      enable tone on this channel
RET
```

It is often possible to get a wide range of sound effects simply by altering the Envelope Shape and Envelope Period Registers. You might like to try this. You will find, however, that, just as in BASIC, generating sound effects is a rather "hit and miss" occupation that often requires a little experimentation to get the best results.

Chapter 12

The Amstrad Keyboard

That the keyboard is the main means by which we can communicate with the computer is rather obvious. We type in all our programs on it, and issue all our BASIC commands on it. It would thus be useful if we could somehow access the keyboard from our machine code programs. Well, we can't do it directly, because the effort required to read the information from the Amstrad hardware is considerable, and making use of the results would be a little difficult. Instead, we'll choose the easy option; that is, use the facilities that are offered by the Amstrad ROM. There are several different routines in the ROM that are used to read the keyboard and make sense of the values read from the hardware. For our purposes, we'll make use of two routines that perform the following tasks.

- (a) Wait until a key is pressed and then return the character typed in a register.
- (b) Read the keyboard, but do not wait until a key is pressed. If the key is being pressed at the instant of reading the keyboard, then it's ASCII code is returned in a register.

Let's now look at these two routines in detail. They are very easy to use.

Wait for a key

This ROM routine is called at address &BB06. Once called it causes the machine to wait until a key is pressed on the keyboard. The key pressed should be one that normally returns a character or the ENTER key. Keys like the SHIFT or SHIFT-LOCK keys will not terminate this routine. Once a key has been pressed, the ASCII code of the character will be passed back from the ROM routine in the CPU A register. Example 23 shows how we might use this call.

Example 23:

The code is shown below. Enter it at address 40000 and use the BASIC program to demonstrate it.

```

LD   L,(IX+0)
LD   H,(IX+1)
CALL &BB06
LD   (HL),A
RET

100 A%=0
110 CALL 40000,@A%
120 PRINT A%,CHR$(A%)
130 GOTO 110

```

The character code is thus returned in the variable A%.

This routine wouldn't be of much use in most games programs, for as soon as you stopped pressing keys the action in the game would grind to a halt until a key was pressed. However, there are several applications where this call is useful, in BASIC as well as machine code programs.

The first is where we want the program to pause until any key is pressed. Calling the above mentioned routine from BASIC will do this admirably. We could even specify the key we want pressing before allowing the user to go on. This requires a short machine code program, and the below routine will cause the computer to wait until the Space Bar has been pressed. Remember that the ASCII code for a 'space' is 32.

```

WAIT   CALL &BB06
        CP   32
        JR   NZ,WAIT
        RET

```

The routine simply causes the ROM routine to be called repeatedly until the Space Bar is pressed. This would cause the ROM routine to be exited with 32 in the A register, and thus in this case the CP 32 instruction would cause the Z flag to be set, thus causing the condition of the JR NZ instruction to fail.

A further application, in machine code programming, might be to wait for a key to be pressed that represents a particular option number from a range of options. The below routine does this, assuming that there are 3 options numbered 1, 2 and 3. The ASCII codes for these numbers are 49,50 and 51.

```

WAIT   CALL &BB06
        CP   49
        JP   Z,OPTION1
        CP   50
        JP   Z,OPTION2
        CP   51
        JP   Z,OPTION3
        JR   WAIT           loop until 1, 2 or 3
OPTION1 ...                is pressed

```

You could write a short machine code program that enters a string of characters from the keyboard and stores the ASCII codes of these characters in an area of memory pointed to by an index register. Such a routine is shown below. Pressing the ENTER key will terminate the operation.

```

        LD   IX,41000      address for characters
WAIT    CALL &BB06
        LD   (IX+0),A
        INC IX
        CP   13
        JR   NZ,WAIT
OUT     ...                if CHR#13 go here

```

The thing to note in this program is that the CP 13 instruction is, of course, a CP A,13 instruction in disguise. It does not refer to the contents of the IX register. When I started machine code programming I always made the mistake of assuming that the CP instructions in a situation like this referred to the last register accessed!

Don't Wait For a Key

The second way of reading the keyboard that we're going to look at in this introduction to Amstrad machine code doesn't wait until a key is pressed, but simply goes on with the rest of the program. If a key is pressed the ASCII code of the key is returned in the A register. The ROM routine to perform this function is called at address &BB09. The status of the C flag indicates to us whether a key was pressed in the instant at which the routine was examining the keyboard. If the C flag is set on return from this ROM call, then it indicates that a key was pressed and the the ASCII code of the character hence generated can be found in the A register. If the C flag is clear then it indicates that a key was not pressed during the time that the routine was examining the keyboard. Example 24 shows this routine being called from BASIC.

Example 24:

Enter the machine code at address 40000, and then use the below BASIC program to run it. A% will hold the ASCII code of any character entered during the scan time (when the keyboard was being examined by the ROM routine), or the value 0 if no key was being pressed. Note that the auto-repeat on keys still works when we call these ROM routines.

```

LD     L,(IX+0)
LD     H,(IX+1)
CALL  &BB09
RET

```

The BASIC program is;

```
100 A%=0
110 CALL 40000,@A%
120 PRINT A%
130 GOTO 110
```

You can see that this routine is very useful in machine code games, because processing doesn't stop if you're not pressing a key.

That completes this introduction to the Amstrad keyboard. We'll now go on to look at the other major means of interacting with the computer; the display screen.

Chapter 13

The Amstrad Display

Now we've examined the Sound capabilities of the Amstrad, and seen how we can read information from the keyboard of the computer, we'll complete this introduction to Amstrad machine language by examining how we can interact with the display screen from within our machine code programs. The Amstrad has a high resolution, colour display and most of the facilities that are available from BASIC are also available from machine code with reasonable ease. However, we'll only look at the simpler methods of accessing the Amstrad screen from within our programs. The most obvious thing to do is to find out how we can simply print characters to the screen. Again, due to the complex arrangement of the Amstrad hardware, the job is best done using a ROM routine.

Printing Characters to the Screen

The ROM routine that does this is called at address &BB5A. It's very easy to use; we simply put the ASCII code of the character that we want to print on the screen in the A register, and then we simply call the ROM routine. This routine treats characters passed to it in the A register in two ways, depending upon the ASCII code of the characters.

- (a) Codes between 32 and 255 inclusive are printed on the screen. Thus we can print characters that available from the keyboard as well as user definable characters set up by the SYMBOL command.
- (b) Codes between 0 and 31 are treated in a special fashion by the Amstrad Operating System. These codes are called CONTROL CODES, or CONTROL CHARACTERS.

What are Control Codes?

They provide us with a means of controlling the behaviour of certain aspects of the display. For example, the ENTER key, when pressed, causes two control codes to be sent to the display. These tell the display to move the text cursor to the start of

the next line. Other Control Codes do things like move the text cursor up, down or sideways or they clear the screen, set the PEN colour or make a "beep" noise. We'll look at them in greater detail shortly.

Printing Characters

The below routine will print the character held in the A register to the screen.

```
LD  A,ASCII code
CALL &BB5A
RET
```

As a more useful example, look at the program shown in Example 25.

Example 25:

This program will fill the screen with the character whose ASCII code is specified as the parameter to the CALL instruction that calls the machine code program. Load the bytes to address 40000, and use a command like CALL 40000.A% to run the program.

A% of course, holds the ASCII code of the character of interest. Thus CALL 40000,65 will fill the screen with letter "A"s.

```
LD  A,(IX+0)
LD  BC,1000
LOOP  PUSH AF           preserve the registers
      PUSH BC
      CALL &BB5A
      POP BC
      DEC BC
      LD  A,B
      OR  C
      JR  Z,OUT
      POP AF
      JR  LOOP
OUT   POP AF
      RET
```

Let's look at the program; the first instruction first recovers the ASCII code from the parameter block. The LD BC instruction then loads this register pair with the number of text character locations on the screen. This is 1000 in mode 1, and is different in the other two modes. You can modify this program to fill screens in the other modes by simply modifying the number loaded into the BC register pair. The number of characters that can be put on the screen in a given mode is given by $V*H$, where V is the number of screen lines and H is the number of columns on the screen.

The registers are pushed onto the stack because the ROM routines often mess up the contents of register pairs.

We can also print out strings of characters from machine code. These strings could, for example, represent messages that the machine code programs should print out while they are running. It's very easy to do this, as Example 26 shows.

Example 26:

This routine prints the message "Hello" to the screen, a rather trivial application! However, the principles are applicable to longer strings, and all that needs to be changed in the program is the value loaded into the IX register as the start address of the string of characters and the value loaded into the B register as the length of the string of characters to be printed. The below line of BASIC can be used to put the string into memory at address 41000.

```
FOR I=0 TO LEN("Hello"):
POKE (41000+I),ASC(MID$("Hello",I,1)):NEXT I
```

The below bytes should be loaded to address 40000. CALL 40000 will then print the string.

```
LD IX,41000      address of string
LD B,5          length of string
LOOP LD A,(IX)
PUSH BC        save no. of characters
CALL &BB5A
POP BC
INC IX
DEC B
JR NZ,LOOP
RET
```

It's all very well to be able to print a string of characters to the screen, but in BASIC we are also able to specify exactly where on the screen the characters are to be placed. Can we do this in machine code?

The answer is yes.

Positioning Text on the Screen

We use one of the control codes to do this. If we send character 31 to the ROM routine at &BB5A, then the next two numbers to be passed to the routine are not treated as characters to be printed to the screen. Instead, they are treated as the X and Y position at which the next characters to be printed are to be placed. Thus, if we were to send the numbers 31, 10, 12, 65 to the ROM routine, we'd get the letter "A"

printed at position 10, 12 on the screen. In machine code this is

```
LD   A,31
CALL &BB5A      send character 31
LD   A,10
CALL &BB5A      send the X coordinate
LD   A,12
CALL &BB5A      send the Y coordinate
LD   A,65
CALL &BB5A      send the character
```

The X coordinate is between 1 and 80 in value, and specifies the column number on the screen. The maximum value here that will give a sensible result depends upon the screen mode in use. For mode 0 the maximum value is 20, in mode 1 it is 40 and in mode 2 it is 80. Should you exceed this value, no error message is given but the character concerned will not be printed to the appropriate position on the screen. Column 1 is the leftmost column of the screen, and the X coordinate increases from left to right. The Y coordinate refers to the screen line number at which you want the character to appear, and varies from 1 to 25 in all modes. Line 1 is the top line of the display.

Thus by using CHR\$31 and it's parameters you can position text at any position on the screen. You can thus see that it provides us with the machine code equivalent of the BASIC LOCATE command. Now we've met a control code, let's take a closer look at some of the others that are available.

Make A 'beep'

For those occasions where you just want a brief tone to indicate that something in your program has occurred, but you don't want to have to program the various registers of the PSG to get the desired effect, you might like to try the below routine, which uses CHR\$7.

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

Clear the Screen

Printing CHR\$12 will clear the screen to the currently selected text paper colour. It is thus equivalent to the BASIC CLS command.

Set the PEN and PAPER Colours

The colour in which text is written to the text cursor on the screen is specified by the PEN command in BASIC.


```
PEN      n
```

will set the colour, where n is the colour number required. To simulate this from machine code, we use CHR\$15. This, followed by a second number, selects the colour specified by the second number. Thus to execute a

```
PEN      2
```

command from within a machine code program, we might use

```
LD      A,15
CALL    &BB5A
LD      A,2
CALL    &BB5A
RET
```

We can do a similar thing to change the currently selected paper. Here we use CHR\$14 as the control code. The below routine executes the PAPER 1 command followed by a CLS. This sets the text screen to the colour specified by the PAPER command.

```
LD      A,14
CALL    &BB5A
LD      A,1
CALL    &BB5A
LD      A,12
CALL    &BB5A
RET
```

There are other control codes available, to enable us to simulate the INK, SYMBOL and MODE commands, but these are often done just as easily from BASIC. One thing to note about the sending of control codes to the display routines is that you should always ensure that the appropriate number of parameters required by that control code are passed as well. If you don't do this, then occasionally strange results can be seen on the screen, as in the absence of parameters the next few character codes are treated as parameters for the control code.

To make things more confusing, all the ASCII codes between 0 and 31 also have a character associated with them. For example, CHR\$7, as well as producing a "beep" noise, can also print a small "space invader" style character to the screen. The problem is, how do we get the ROM routines to print these additional characters to the screen instead of treating the ASCII codes as Control Codes?

The answer is very easy. The control code represented by CHR\$1 causes the code immediately following it to be treated as a printable character. Thus, sending 1 and 7 to the ROM routine in that order

will print the "space invader" to the screen instead of making a "beep". The below machine code instructions will do this.

```
LD    A,1
CALL  &BB5A
LD    A,7
CALL  &BB5A
RET
```

The influence of CHR\$1 only extends over the character immediately following it to the ROM routine. Thus the below program

```
LD    A,1
CALL  &BB5A
LD    A,7
CALL  &BB5A
LS    A,7
CALL  &BB5A
RET
```

will print 1 character to the screen and then generate 1 "beep".

Of course, as well as printing textual information to the screen we also can draw graphics on the screen from BASIC. As you might expect, there are several ROM routines that enable the user to access graphics from machine code routines. Let's now go and look at some of the simpler ROM routines that are available.

Simple Machine Code Graphics

On any microcomputer, the generation of graphics from within machine code programs is quite a job, and so here I'll just give an introduction to the techniques that are used on the Amstrad computer. Again, the job is made easier by the use of the built in ROM routines; accessing the screen directly, by writing bytes to the memory addresses at which the screen is situated is quite difficult because of the complex arrangement of the screen.

The simplest thing that we can do in graphics programming is to move the graphics cursor around on the screen. This is easily done using a ROM routine that we call at address &BBC0. For example, the below routine will move the graphics cursor to coordinate 100,100 on the screen.

```
LD    DE,100      X coordinate
LD    HL,100      Y coordinate
CALL  &BBC0
RET
```

The X and Y coordinates are passed to this ROM routine in the DE and HL register pairs respectively. This

routine appears to mess up the CPU registers, and so it's often useful to PUSH any registers that we're concerned about on to the stack before calling the ROM routine.

The above routine will not put anything on the screen; it simply moves the graphics cursor to the required point on the screen. As an example of its use, we'll call upon the services of another control code, CHR\$5. This enables us to print a text character at the position of the graphics cursor, rather than at the current position of the text cursor. The character that is sent to the "print a character" ROM routine immediately after the CHR\$5 is printed at the graphics cursor. Example 27 shows this in action.

Example 27:

Load the machine code to address 40000, and then call it in the below fashion.

```
CALL 40000
```

This will then print the letter "A" at graphics coordinate 100,100. The character is printed in such a way that the top left corner of the character grid is situated at the graphics cursor position. The character is printed in the current graphics ink colour, rather than the current text ink colour. The machine code is;

```
LD HL,100
LD DE,100
CALL &BBC0
LD A,5
CALL &BBSA
LD A,65
CALL &BBSA
RET
```

This facility is quite useful in that it enables us to position text on the screen to a much greater degree of precision than by using the text coordinate system. As in BASIC, the coordinates for graphics normally have their origin at the bottom left corner of the screen.

The other simple graphics operation that we'll look at is the action of drawing a line on the screen from one point to another point. Again, there's a ROM routine to do all the hard work for us. We use it in the below fashion, by putting the X coordinate into the DE register pair and the Y coordinate of the point to be drawn to in the HL register pair.

```
LD DE,200      X coordinate
LD HL,200      Y coordinate
CALL &BBF6
```

There will draw a line, in the current graphics ink, from the current position of the graphics cursor to the point specified in the DE and HL registers above. Example 28 shows a simple routine for drawing a box on the screen.

Example 28:

Load the code to address 40000, and run it with CALL 40000.

```
LD   DE,100
LD   HL,100
CALL &BBC0      move to point 100,100
LD   DE,100
LD   HL,200
CALL &BBF6      draw to 100,200
LD   DE,200
LD   HL,200
CALL &BBF6      draw to 200,200
LD   DE,200
LD   HL,100
CALL &BBF6      draw to 200,100
LD   DE,100
LD   HL,100
CALL &BBF6      draw to 100,100
RET
```

As you can see, drawing a line is a very easy job. We can also simulate a PLOT statement from our machine code programs using the routines that we've already seen. Simply put, when we plot a point on the screen we are effectively drawing a very short line. So, we should be able to do such a job with the move and draw ROM routines that we've seen. Example 29 shows a simple point plotting routine.

Example 29:

Load the bytes to address 40000, and then execute the program with a CALL 40000,X%,Y% call where X% and Y% represent the X and Y coordinates of the point to be plotted. Although the BASIC contains a PLOT command, this is a useful demonstration of the graphics calls.

```

LD    L,(IX+0)    get the Y coordinate
LD    H,(IX+1)
LD    E,(IX+2)    get the X coordinate
LD    D,(IX+3)
PUSH  HL
PUSH  DE
CALL  &BBC0       do the move operation
POP   DE
POP   HL
INC   DE
INC   DE
CALL  &BBF6       increment DE to give a small X
RET                                coordinate increase and draw line

```

Thus, the call

```
CALL 40000,100,100
```

will plot a point at coordinate 100,100 on the screen, again in the current graphics ink.

One really big advantage that we can get from using ROM routines is that they work equally well in all screen modes. If we were to write screen handling routines of our own, we'd have to take the different screen modes into account.

This is as far as we go in this introduction to Amstrad machine code. Hopefully, you've now past the "absolute beginner" stage and are ready to build on your new found skills. You will no doubt find many articles or books that will expand upon what I've mentioned in this book. The only recommendation that I'll make is that you obtain, if possible, a rather weighty but extremely useful book published by Amsoft called "The Amstrad Firmware Technical Manual" which documents in detail all the many ROM calls of the Amstrad Operating System. Good luck with your machine code programming, and may all your crashes be little ones!

Appendix 1

Instructions and Op-codes

MNEMONIC	HEXADECIMAL	MNEMONIC	HEXADECIMAL	MNEMONIC	HEXADECIMAL
ADC A, (HL)	8E	BIT 2,B	CB 50	CP n	FE XX
ADC A, (IX+dis)	DD BE XX	BIT 2,C	CB 51	CP E	8B
ADC A, (IY+dis)	FD BE xx	BIT 2,D	CB 52	CP H	8C
ADC A,A	8F	BIT 2,E	CB 53	CP L	8D
ADC A,B	88	BIT 2,H	CB 54	CPD	ED A9
ADC A,C	89	BIT 2,L	CB 55	CPDR	ED B9
ADC A,D	8A	BIT 3,(HL)	CB 5E	CPI	ED A1
ADC A,n	CE XX	BIT 3,(IX+dis)	DD CB XX 5E	CPIR	ED B1
ADC A,E	8B	BIT 3,(IY+dis)	FD CB XX 5E	CPL	2F
ADC A,H	8C	BIT 3,A	CB 5F	DAA	27
ADC A,L	8D	BIT 3,B	CB 58	DEC (HL)	35
ADC HL,BC	ED 4A	BIT 3,C	CB 59	DEC (IX+dis)	DD 35 XX
ADC HL,DE	ED 5A	BIT 3,D	CB 5A	DEC (IY+dis)	FD 35 XX
ADC HL,HL	ED 6A	BIT 3,E	CB 5B	DEC A	3D
ADC HL,SP	ED 7A	BIT 3,H	CB 5C	DEC B	05
ADD A, (HL)	86	BIT 3,L	CB 5D	DEC BC	0B
ADD A, (IX+dis)	DD 86XX	BIT 4,(HL)	CB 66	DEC C	0D
ADD A, (IY+dis)	FD 86XX	BIT 4,(IX+dis)	DD CB XX 66	DEC D	15
ADD A,A	87	BIT 4,(IY+dis)	FD CB XX 66	DEC DE	1B
ADD A,B	80	BIT 4,A	CB 67	DEC E	1D
ADD A,C	81	BIT 4,B	CB 60	DEC H	25
ADD A,D	82	BIT 4,C	CB 61	DEC HL	2B
ADD A,n	C6 XX	BIT 4,D	CB 62	DEC IX	DD 2B
ADD A,E	83	BIT 4,E	CB 63	DEC IY	FD 2B
ADD A,H	84	BIT 4,H	CB 64	DEC L	2D
ADD A,L	85	BIT 4,L	CB 65	DEC SP	3B
ADD HL,BC	09	BIT 5,(HL)	CB 6E	DI	F3
ADD HL,DE	19	BIT 5,(IX+dis)	DD CB XX 6E	DJNZ,dis	10 XX
ADD HL,HL	29	BIT 5,(IY+dis)	FD CB XX 6E	EI	FB
ADD HL,SP	39	BIT 5,A	CB 6F	EX (SP),HL	E3
ADD IX,BC	DD 09	BIT 5,B	CB 68	EX (SP),IX	DD E3
ADD IX,DE	DD 19	BIT 5,C	CB 69	EX (SP),IY	FD E3
ADD IX,IX	DD 29	BIT 5,D	CB 6A	EX AF,AF'	08
ADD IX,SP	DD 39	BIT 5,E	CB 6B	EX DE,HL	EB
ADD IY,BC	FD 09	BIT 5,H	CB 6C	EXX	D9
ADD IY,DE	FD 19	BIT 5,L	CB 6D	HALT	76
ADD IY,IX	FD 29	BIT 6,(HL)	CB 76	IM 0	ED 46
ADD IY,SP	FD 39	BIT 6,(IX+dis)	DD CB XX 76	IM 1	ED 56
AND (HL)	A6	BIT 6,(IY+dis)	FD CB XX 76	IM 2	ED 5E
AND (IX+dis)	DD A6 XX	BIT 6,A	CB 77	IN A, (C)	ED 78
AND (IY+dis)	FD A6 XX	BIT 6,B	CB 70	IN A,port	DB XX
AND A	A7	BIT 6,C	CB 71	IN B, (C)	ED 40
AND B	A0	BIT 6,D	CB 72	IN C, (C)	ED 48
AND C	A1	BIT 6,E	CB 73	IN D, (C)	ED 50
AND D	A2	BIT 6,H	CB 74	IN E, (C)	ED 58
AND n	E6 XX	BIT 6,L	CB 75	IN H, (C)	ED 60
AND E	A3	BIT 7,(HL)	CB 7E	IN L, (C)	ED 68
AND H	A4	BIT 7,(IX+dis)	DD CB XX 7E	INC (HL)	34
AND L	A5	BIT 7,(IY+dis)	FD CB XX 7E	INC (IX+dis)	DD 34 XX
BIT 0,(HL)	CB 46	BIT 7,A	CB 7F	INC (IY+dis)	FD 34 XX
BIT 0,(IX+dis)	DD CB XX 46	BIT 7,B	CB 78	INC A	3C
BIT 0,(IY+dis)	FD CB XX 46	BIT 7,C	CB 79	INC B	04
BIT 0,A	CB 47	BIT 7,D	CB 7A	INC BC	03
BIT 0,B	CB 40	BIT 7,E	CB 7B	INC C	0C
BIT 0,C	CB 41	BIT 7,H	CB 7C	INC D	14
BIT 0,D	CB 42	BIT 7,L	CB 7D	INC DE	13
BIT 0,E	CB 43	CALL ADDR	CD XX XX	INC E	1C
BIT 0,H	CB 44	CALL C,ADDR	DC XX XX	INC H	24
BIT 0,L	CB 45	CALL M,ADDR	FC XX XX	INC HL	23
BIT 1,(HL)	CB 4E	CALL NC,ADDR	D4 XX XX	INC IX	DD 23
BIT 1,(IX+dis)	DD CB XX 4E	CALL NZ,ADDR	C4 XX XX	INC IY	FD 23
BIT 1,(IY+dis)	FD CB XX 4E	CALL P,ADDR	F4 XX XX	INC L	2C
BIT 1,A	CB 4F	CALL PE,ADDR	EC XX XX	INC SP	33
BIT 1,B	CB 48	CALL PO,ADDR	E4 XX XX	IND	ED AA
BIT 1,C	CB 49	CALL Z,ADDR	CC XX XX	INCR	ED BA
BIT 1,D	CB 4A	CCF	3F	INI	ED A2
BIT 1,E	CB 4B	CP (HL)	BE	INIR	ED B2
BIT 1,H	CB 4C	CP (IX+dis)	DD BE XX	JP (HL)	E9
BIT 1,L	CB 4D	CP (IY+dis)	FD BE XX	JP (IX)	DD E9
BIT 2,(HL)	CB 56	CP A	BF	JP (IY)	FD E9
BIT 2,(IX+dis)	DD CB XX 56	CP B	B8	JP ADDR	C3 XX XX
BIT 2,(IY+dis)	FD CB XX 56	CP C	B9	JP C,ADDR	DA XX XX
BIT 2,A	CB 57	CP D	BA	JP M,ADDR	FA XX XX

MNEMONIC	HEXADECIMAL	MNEMONIC	HEXADECIMAL	MNEMONIC	HEXADECIMAL
JP NC,ADDR	D2 XX XX	LD BC,nn	01 XX XX	LDDR	ED B8
JP NZ,ADDR	C2 XX XX	LD C, (HL)	4E	LDI	ED A0
JP P,ADDR	F2 XX XX	LD C, (IX+dis)	DD 4E xx	LDIR	ED 80
JP PE,ADDR	EA XX XX	LD C, (IY+dis)	FD 4E XX	NEG	ED 44
JP PO,ADDR	E2 XX XX	LD C,A	4F	NOP	00
JP Z,ADDR	CA XX XX	LD C,B	48	OR (HL)	B6
JR C,dis	38 XX	LD C,C	49	OR (IX+dis)	DD B6 XX
JR dis	18 XX	LD C,D	4A	OR (IY+dis)	FD B6 xx
JR NC,dis	30 XX	LD C,n	0E XX	OR A	B7
JR NZ,dis	20 XX	LD C,E	4B	OR B	B0
JR Z,dis	28 XX	LD C,H	4C	OR C	B1
LD (ADDR) ,A	32 XX XX	LD C,L	4D	OR D	B2
LD (ADDR) ,BC	ED 43 XX XX	LD D, (HL)	56	OR n	F6 XX
LD (ADDR) ,DE	ED 53 XX XX	LD D, (IX+dis)	DD 56 XX	OR E	B3
LD (ADDR) ,HL	ED 63 XX XX	LD D, (IY+dis)	FD 56 XX	OR H	B4
LD (ADDR) ,HL	22 XX XX	LD D,A	57	OR L	B5
LD (ADDR) ,IX	DD 22 XX XX	LD D,B	50	OTDR	ED 8B
LD (ADDR) ,IY	FD 22 XX XX	LD D,C	51	OTIR	ED 83
LD (ADDR) ,SP	ED 73 XX XX	LD D,D	52	OUT (C) ,A	ED 79
LD (BC) ,A	02	LD D,n	16 XX	OUT (C) ,B	ED 41
LD (DE) ,A	12	LD D,E	53	OUT (C) ,C	ED 49
LD (HL) ,A	77	LD D,H	54	OUT (C) ,D	ED 51
LD (HL) ,B	70	LD D,L	55	OUT (C) ,E	ED 59
LD (HL) ,C	72	LD DE, (ADDR)	ED 5B XX XX	OUT (C) ,H	ED 61
LD (HL) ,D	71	LD DE,nn	11 XX XX	OUT (C) ,L	ED 69
LD (HL) ,n	36 XX	LD E, (HL)	5E	OUT part,A	D3 port
LD (HL) ,E	73	LD E, (IX+dis)	DD 5E XX	OUTD	ED AB
LD (HL) ,H	74	LD E, (IY+dis)	FD 5E XX	OUTI	ED A3
LD (HL) ,L	75	LD E,A	5F	POP AF	F1
LD (IX+dis) ,A	DD 77 XX	LD E,B	58	POP BC	C1
LD (IX+dis) ,B	DD 70 XX	LD E,C	59	POP DE	D1
LD (IX+dis) ,C	DD 71 XX	LD E,D	5A	POP HL	E1
LD (IX+dis) ,D	DD 72 XX	LD E,n	1E XX	POP IX	DD E1
LD (IX+dis) ,n	DD 36 XX XX	LD E,E	5B	POP IY	FD E1
LD (IX+dis) ,E	DD 73 XX	LD E,H	5C	PUSH AF	F5
LD (IX+dis) ,H	DD 74 XX	LD E,L	5D	PUSH BC	C5
LD (IX+dis) ,L	DD 75 XX	LD H, (HL)	66	PUSH DE	D5
LD (IY+dis) ,A	FD 77 XX	LD H, (IX+dis)	DD 66 XX	PUSH HL	E5
LD (IY+dis) ,B	FD 70 XX	LD H, (IY+dis)	FD 66 XX	PUSH IX	DD E5
LD (IY+dis) ,C	FD 71 XX	LD H,A	67	PUSH IY	FD E5
LD (IY+dis) ,D	FD 72 XX	LD H,B	60	RES 0, (HL)	CB 86
LD (IY+dis) ,n	FD 36 XX XX	LD H,C	61	RES 0, (IX+dis)	DD CB XX 86
LD (IY+dis) ,E	FD 73 XX	LD H,D	62	RES 0, (IY+dis)	FD CB XX 86
LD (IY+dis) ,H	FD 74 XX	LD H,n	26 XX	RES 0,A	CB 87
LD (IY+dis) ,L	FD 75 XX	LD H,E	63	RES 0,B	CB 80
LD A, (ADDR)	3A XX XX	LD H,H	64	RES 0,C	CB 81
LD A, (BC)	0A	LD H,L	65	RES 0,D	CB 82
LD A, (DE)	1A	LD HL, (ADDR)	ED 68 XX XX	RES 0,E	CB 83
LD A, (HL)	7E	LD HL, (ADDR)	2A XX XX	RES 0,H	CB 84
LD A, (IX+dis)	DD 7E XX	LD HL,nn	21 XX XX	RES 0,L	CB 85
LD A, (IY+dis)	FD 7E XX	LD I,A	ED 47	RES 1, (HL)	CB 8E
LD A,A	7F	LD IX, (ADDR)	DD 2A XX XX	RES 1, (IX+dis)	DD CB XX 8E
LD A,B	78	LD IX,nn	DD 21 XX XX	RES 1, (IY+dis)	FD CB XX 8E
LD A,C	79	LD IY (ADDR)	FD 2A XX XX	RES 1,A	CB 8F
LD A,D	7A	LD IY,nn	FD 21 XX XX	RES 1,B	CB 88
LD A,n	3E XX	LD L,A	6F	RES 1,C	CB 89
LD A,E	7B	LD L,B	68	RES 1,D	CB 8A
LD A,H	7C	LD L,C	69	RES 1,E	CB 8B
LD A,I	ED 57	LD L,D	6A	RES 1,H	CB 8C
LD A,L	7D	LD L,n	2E XX	RES 1,L	CB 8D
LD A,R	ED 5F	LD L,E	6B	RES 2, (HL)	CB 96
LD B, (HL)	46	LD L, (HL)	6E	RES 2, (IX+dis)	DD CB XX 96
LD B, (IX+dis)	DD 46 XX	LD L, (IX+dis)	DD 6E XX	RES 2, (IY+dis)	FD CB XX 96
LD B, (IY+dis)	FD 46 XX	LD L, (IY+dis)	FD 6E XX	RES 2,A	CB 97
LD B,A	47	LD L,H	6C	RES 2,B	CB 90
LD B,B	40	LD L,L	6D	RES 2,C	CB 91
LD B,C	41	LD R,A	ED 4F	RES 2,D	CB 92
LD B,D	42	LD SP, (ADDR)	ED 7B XX XX	RES 2,E	CB 93
LD B,n	06 XX	LD SP,nn	F1 XX XX	RES 2,H	CB 94
LD B,E	43	LD SP,HL	F9	RES 2,L	CB 95
LD B,H	44	LD SP,IX	DD F9	RES 3, (HL)	CB 9E
LD B,L	45	LD SP,IY	FD F9	RES 3, (IX+dis)	DD CB XX 9E
LD BC, (ADDR)	ED 4B XX XX	LDD	ED A8	RES 3, (IY+dis)	FD CB XX 9E
				RES 3,A	CB 9F

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

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

Appendix 2

Flag Operation Summary

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

SYMBOL	OPERATION
C	Carry flag. C=1 if the operation produced a carry from the most significant bit of the operand or result.
Z	Zero flag. Z=1 if the result of the operation is zero.
S	Sign flag. S=1 if the most significant bit of the result is one, ie a negative number.
P/V	Parity or overflow flag. Parity (P) and overflow (O) share the same flag. Logical operations affect this flag with the parity of the result while arithmetic operations affect this flag with the overflow of the result. If P/V holds parity, P/V=1 if the result of the operation is even, P/V=0 if result is odd. If P/V holds overflow, P/V=1 if the result of the operation produced an overflow.
H	Half-carry flag. H=1 if the add or subtract operation produced a carry into or borrow from bit 4 of the accumulator.
N	Add/Subtract flag. N=1 if the previous operations was a subtract.
	H and N flags are used in conjunction with the decimal adjust instruction (DAA) to properly correct the result into packed BCD format following addition or subtraction using operands with packed BCD format.
#	The flag is affected according to the result of the operation.
-	The flag is unchanged by the operation.
0	The flag is reset (=0) by the operation.
1	The flag is set (=1) by the operation.
X	The flag result is unknown.
V	The P/V flag is affected according to the overflow result of the operation.
P	P/V flag is affected according to the parity result of the operation.
r	Any one of the CPU registers A,B,C,D,E,H,L.
s	Any 8-bit location for all the addressing modes allowed for the particular instructions.
SS	Any 16-bit location for all the addressing modes allowed for that instruction.
R	Refresh register
n	8-bit value in range 0-255.
nn	16-bit value in range 0-65535.

Appendix 3

Numbers on the Amstrad

In this Appendix I want to take a brief look at the way in which the Amstrad allows us to convert from one number representation to another.

There are a couple of built in functions in the Amstrad that allow us to convert numbers from decimal to both binary and hexadecimal. Let's look at these.

BIN\$ (value, no. of digits)

This function accepts a decimal number as 'value' and returns a string representing the binary representation of the number. The 'no of digits' parameter allows us to specify how many binary digits are returned when we use the function. This parameter is optional, but is very useful. It makes sure that a value is returned with leading zeros where they are needed. To make this clearer;

```
PRINT BIN$(8) will print 1000
PRINT BIN$(8,8) will print 00001000
```

which is closer to the way in which we are used to seeing binary numbers, as the contents of a byte or register.

HEX\$ (value)

This function returns the hexadecimal representation of the decimal number 'value'. For example,

```
PRINT HEX$(65536) will print FFFF
```

This function is really useful when we want to work out the displacement bytes for relative jumps or index register operations. Thus, if we know that the displacement value for a relative jump is -2, and we want to get the hexadecimal representation of this value using Two's Complement notation, we simply use

```
PRINT HEX$(-2)
```

The answer, FFFF, is converted into a single byte value suitable for inclusion in our programs by simply

discarding the first two digits, thus leaving us with FE.

We can, of course, incorporate hexadecimal or binary numbers into our programs using the appropriate prefix. This is '&' for hexadecimal numbers and '&X' for binary numbers. Thus, 3 is represented in 8 digit binary as

&X00000011

and in hexadecimal as

&3

Appendix 4

Timing Programs

It is possible to work out, roughly, the length of time a given machine code program will take to run. The Z80 CPU in the Amstrad is given about 3.3 million "ticks" of it's internal clock each second. Thus, one tick occupies about 0.33 millionths of a second. Now, in the tables of instructions that we've seen scattered through this book the timings have been quoted in terms of these ticks. The absolute length of time taken for the CPU to execute a given instruction is thus

$$\text{TIME} = 0.33 * \text{number of ticks}$$

where TIME will be in millionths of a second, or micro seconds.

When we use this technique in machine code programs, remember that instructions in loops will be repeated several times, and also remember that some conditional instructions take different numbers of "ticks" depending upon whether the condition is fulfilled or not.

Index

@	47,49	Labels	42
Accumulator	10,63,68	LD	35-45
ADC	63-66,70,90-92	Loading Files	33
ADD	63-66,70,90-92	Logical Operations	70-76
Addressing	7	LSB	19
Addressing Modes	35	Machine Code	1-2
Alternative Registers	10,85	Machine Language	1-2
Amplitude	121	Memory	12,25-29
Amplitude Control Register	121	MEMORY	28
AND	71-72,75	Mnemonic	4
ALU	11	MSB	19
ASCII	24	NEG	116
Assembly Language	3-5	Nibble	19
BASIC	1,25-34	Noise	124-125
BCD	67-68	NOP	113
Beep	138	NOT	71
Binary	16,51	Op Code	37
Bits	19	Operand	37
Block Operations	106-110	Operating System	2
Bytes	19-20	OR	73-74,75
CALL (in BASIC)	30-32,47-53	OUT	111-113
CALL (in machine code)	101-104	Overflow	57-58
Channels	117	Parameters	31,47-53
Characters	24,135-136	Parity	57
Clear (of flags)	55	PEEK	30
Clear Screen	138	PEN	138-139
CLR (key)	122	POKE	29-30
Coarse Tune Control	119	POP	84-86
Compare Instructions	68-70,106-108	Positioning Text	137
Complement	23	PPI	13
Conditional Jumps	94	Printing Characters	136
Control Codes	135	Program	21
Control Unit	11	Program Counter	11
Counting	15-24,55-70,89-92	PSG	13,117-129
CPU	1,5-7,8-12	PUSH	84-86
CRTC	13	RAM	12
DAA	68	Real Numbers	47
Data	21	Registers	8,10,117
Decrement	61-62,89-90	Register Addressing	36
Destination	35	Register Indirect Addressing	37
Displacement	96-98	RESET (RES)	55,75-76
Display	135-143	Restarts	104
DJNZ	99-100	RET	34,104
Envelopes	126-129	RLD	114-115
Extended Addressing	39	ROM	12
F Register	8,10,55-59	Rotates	76-79
Fine Tune Control Register	119	Running Tape Files	32
Flags	55-59	Saving Files	32
Gate Array	13	Saving Registers	102
Graphics	140-143	SBC	66-67,70,91-92
HALT	115	SET	75,76
Hand Assembly	4	Set	55
Hexadecimal	17-19,151	Shifts	76-79
HIMEM	27-29,33	Signed Integers	21-22
Homes for Machine Code	25-29	Source Register	35
Immediate Addressing	36-37	Stack	6,83-85
Immediate Indexed Addressing	43-44	Stack Pointer	11,83-85
IN	111-113	Strings	24,51
Increment	59-62,89-90	String Descriptor	51-52
Indexed Addressing	42	SUB	66-67,70,91-92
Instruction Code	37	Timing Programs	153
Instruction Register	11	Tone Envelope	124
Integer	21-24,48	Truth Tables	71
Interrupt	105	Two's Complement	21-24
Interrupt Modes	115-116	Unconditional Jumps	93-94
Jumps		XOR	74,75
Absolute	93-96	Z80	1,5,8-12
Register Indirect	98	Zero Flag	58
Relative	96-100		
Keyboard	131-134		

Amstrad

CPC 464



Melbourne
House

This book will enable you to learn machine language the easy way — no computer jargon. A straight forward approach with many examples.

Compiled exclusively for Amstrad users, **AMSTRAD MACHINE LANGUAGE FOR THE ABSOLUTE BEGINNER** offers complete instructions in Z80 machine language programming.

If you are frustrated by the limitations of BASIC and want to write faster, more powerful, space-saving programs or subroutines, then this book is for you.

Even with no previous experience of computer languages, the easy-to-understand 'no jargon' format of this book will enable you to discover the power of the Amstrad's own language.

Each chapter includes specific examples of machine language applications which can be demonstrated and used on your own Amstrad. The features and capabilities of the Amstrad are all covered, so you can start programming straight away.

AMSTRAD MACHINE LANGUAGE FOR THE ABSOLUTE BEGINNER takes you, in logical steps, through a comprehensive course in machine language programming. This book gives you everything you need to write machine language programs on your Amstrad.



Melbourne
House
Publishers

ISBN 0-86161-193-4



9 780861 611935

FORGOTTEN

AND
RECOVERED

MEMORIES

OF
THE
PAST

AND
THE
FUTURE



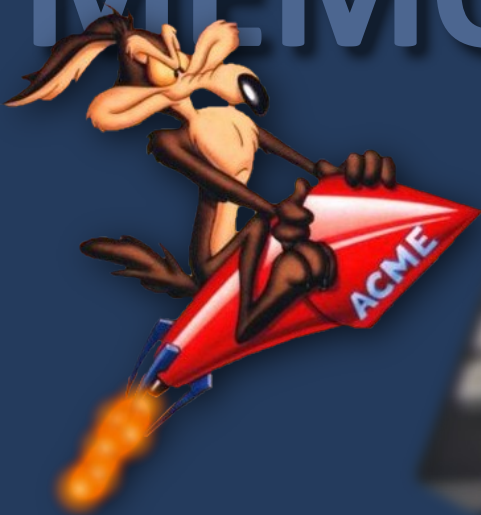


Document **numérisé**
avec amour par :

AMSTRAD

CPC 

MÉMOIRE ÉCRITE



<https://acpc.me/>