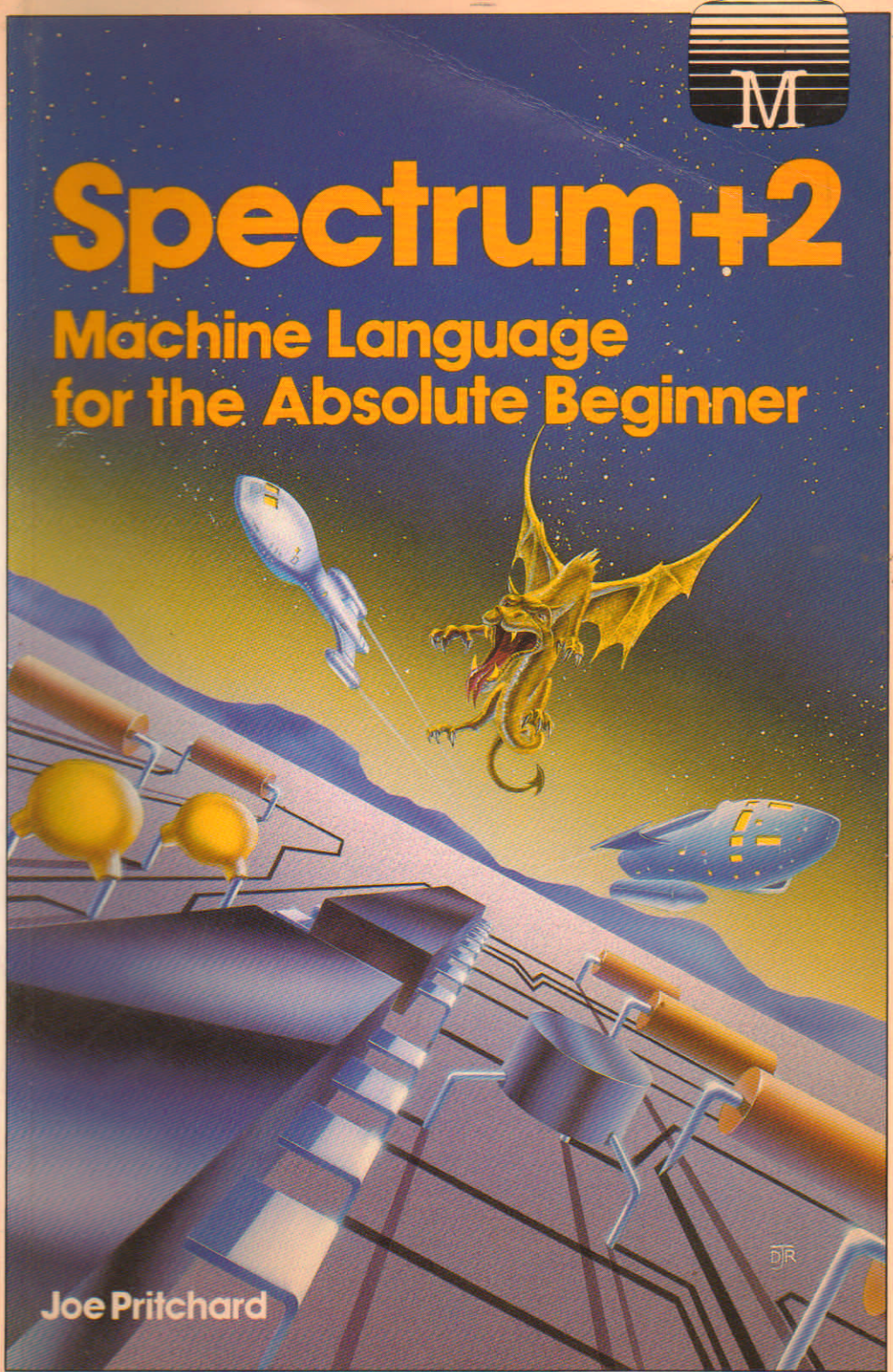
Spectrum+2

Machine Language for the Absolute Beginner



Joe Pritchard

Spectrum+2

Machine Language for the Absolute Beginner

Joe Pritchard

MELBOURNE HOUSE
PUBLISHERS

Published in United Kingdom by
Melbourne House (Publishers) Ltd
60 High Street, Hampton Wick
Kingston-Upon-Thames
Surrey KT1 4DB

Published in Australia by
Melbourne House (Australia) Pty Ltd
96-100 Tope Street
South Melbourne, Victoria 3205

ISBN 0 86161 209 4

Printed and bound in Great Britain by Short Run Press Ltd., Exeter.

Contents

Chapter 1
Machine Code First Principles

This book is designed to introduce the Spectrum 128/Plus 2 BASIC programmer to the 'mother tongue' of his or her computer. You may have previously heard of machine language, or it may be totally new to you. Whichever category you fall into, don't worry; we'll start our investigation of machine language, or **MACHINE CODE**, as it's sometimes called, by looking in this Chapter at the basic principles of the subject.

The first thing to do is to look at the way in which we normally program our computer. We type in a line of BASIC and then it is 'vetted' by the Spectrum ROM to see if it's legal; this is the same whether you're working in 48 or 128 BASIC. (As an aside, unless I otherwise mention it the machine will be running in 128 mode.) If the line is OK, it's entered into the program or executed, depending upon whether we gave it a line number or not.

When we're typing instructions into the computer in this way, we're not really communicating with the 'brain' of the computer, the **Central Processor Unit**, or the **CPU**. When we program the computer in BASIC, we never actually access the CPU directly; instead we go through a 'middle man' called the BASIC Interpreter, which directly accesses the CPU.

The CPU used in the 128/Plus 2 is called the Z80, and is probably the most popular CPU around in home computers at the moment. It was used in the predecessors of the 128/Plus 2, the 48k Spectrum and the Spectrum+, and has also been used in the other popular Amstrad machines. The CPU is an electronic 'chip', and is the heart of all the activity of the computer. Other chips are to be found in the 128, but they are all under the direct or indirect control of the CPU. When we talk about programming the 128/Plus 2 in machine code, we're actually talking about programming the Z80 CPU of the computer in machine code.

Why Bother?

Spectrum Plus 2 BASIC is a reasonably good version of BASIC; it's a little slow, but for many programs it is quite adequate. Why should we bother learning a new language? Well, there are three main advantages that machine code offers us over BASIC.

1. Faster programs.
2. Programs written in machine code can use less memory to do the same job.
3. Certain tasks can ONLY be done using machine code.

In addition, machine code allows us to free ourselves from the limitations of BASIC, and can also allow us to alter the way in which BASIC works! Thus, a knowledge of machine code can be a very useful thing. Having hopefully answered the question of 'Why?', let's now look at 'What?', and see exactly what machine code is.

What is Machine Code?

The Z80 CPU, if you've never actually seen one, is a black 'chip', about 2" long, with 40 metal 'legs' on it which form electrical connections between the computer circuit and the silicon chip inside the black plastic case. Of these 'legs', or **PINS**, as they are called, 8 are particularly important. The CPU communicates

with the rest of the computer via electrical signals and the CPU is designed to behave in different ways depending upon the combination of electrical signals on these 8 pins. Remembering that we're talking about electrical signals, let's represent the presence of an electrical signal on one of these pins by a '1' and the absence of an electrical signal as a '0'. As there are 8 pins of particular interest, a typical combination of electrical signals on these pins might be represented as:

00110010

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

We call such a combination of signals a machine language instruction, just as 'LET a=0' is a BASIC instruction. This is essentially what machine code programming is all about; giving the CPU a sequence of combinations of electrical signals that tell it to do a sequence of tasks. The instructions that are understood by the CPU are collectively called the **Z80 INSTRUCTION SET**. Each different CPU has a different instruction set, and so programs written in machine language for one CPU will almost certainly *not* work on a different CPU, because the instruction sets are different.

The BASIC Interpreter

This is a machine language program whose job it is to convert BASIC instructions typed in to the computer into machine language instructions that the CPU can understand. This is similar to the way in which we might use a dictionary to translate from English into a foreign language. Alternatively, if we could afford it, we might use a professional Interpreter.

Part of the BASIC Interpreter in the Spectrum 128 is a series of small programs, similar to BASIC subroutines, that tell the CPU how to communicate with the keyboard, screen, sound chip and tape recorder. In many machines, these routines are totally separate from the BASIC Interpreter, and are called the **OPERATING SYSTEM** (**OS**) of the computer. In

the Spectrum, the BASIC Interpreter and Operating System programs are rather mixed up; I tend to treat all the Operating System machine code programs as being part of the BASIC Interpreter. However, these 'OS' routines of the BASIC Interpreter are called whenever the Interpreter needs to put something on the screen, read the keyboard or send information to tape.

The fact that we have to translate BASIC Instructions to machine code before they can be used is the principle reason for the slowness of BASIC in comparison to machine code programs. The translation process takes time, and also the machine code routines that finally do the required tasks are not as efficient as they could be.

Disadvantages of Machine Code

There are a few disadvantages to using machine code. Just to balance things up a little, let's have a look at them.

1. Machine language programs are difficult to read and find errors in.
2. They are difficult to transfer from one type of computer to another. Usually this transfer involves re-writing parts of the program, if not all of it!
3. Complex arithmetic, such as trigonometry, is difficult using machine code.

You can thus see that 'You pays your money and takes your choice' with regard to whether machine code or BASIC is used to write a particular program. Speed can be provided by machine code, and good mathematical abilities are more easily provided from BASIC. Thus you'd be best to write your epic arcade game in machine code, but that Accounts Program for the office might be easier to write in BASIC.

So far, we've seen that we can represent machine code instructions as a series of 1's and 0's. If we had to write machine code programs like this, then only the masochists of the computing world would bother! It would simply be too time

consuming to write big machine code programs in this way. To us humans, such strings of 1's and 0's can represent a **BINARY NUMBER**, and we can then translate this into decimal. This would be a much more convenient representation of the instructions for us, although the CPU would still see the instruction as a series of electrical signals on those 8 pins. However, we could now write a machine code program as a series of decimal numbers. While this is more intelligible to us than a string of binary numbers, it still doesn't give us any idea of what a particular CPU instruction does. We could have a list of these numbers and the jobs they perform, but this would soon get to be more like doing the yearly accounts than programming a computer! No, what would be useful would be a method of representing machine code instructions in some sort of English. We can, in fact, do this.

We use a form of representation called **ASSEMBLY LANGUAGE**. Each machine code instruction is given a short 'name' which indicates its function to us. This name is called a **MNEMONIC**. (The first 'm' is silent, and the word is pronounced 'nemonic'). An alternative name is an **ASSEMBLER INSTRUCTION**. We now have three different ways of representing instructions to the CPU: binary, decimal or mnemonic. For a particular instruction, let's look at each method.

Binary	01110110
Decimal	118
Mnemonic	HALT

From the mnemonic, you might even be able to guess what this instruction does. Yes, it tells the CPU to stop, or HALT, until further notice.

Of course, such mnemonics are totally incomprehensible to the CPU. We thus require a method of converting mnemonics back into 1s and 0s for the CPU. There are two ways of doing this, known as 'the hard way' and 'the easy way'! The first is to use tables like those in the back of the book to to convert the mnemonics into numbers. This technique is called **HAND ASSEMBLY**, and although it's possible to hand assemble small

programs it's not feasible for larger ones.Though good for the soul, it takes a long time, is prone to errors, and can be very frustrating. The second method is to get a computer program to do the job for you. Much easier!

Such a program is called an **ASSEMBLER**, and it **ASSEMBLES** a machine code program from the mnemonics of the Assembly language program. There are a variety of such programs available for the Spectrum 48 and Spectrum+ and you should be able to find one that works with the Spectrum 128.

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 computer on, the CPU begins running, or **EXECUTING**, the machine code program that initialises the computer and puts the 'main menu' on the screen. This, and all the other jobs done by the CPU may appear to be incredibly complicated, but, in fact, the CPU is only capable of doing very simple tasks, such as addition and moving numbers from one part of the computer to another. However, these operations are carried out very quickly, and it is the speed of operation that gives the CPU its power.

With such simple tasks, the CPU uses what might be called its 'fingers' to carry them out, just as a child might when performing simple sums. It can make use of 'pencil and paper', in order to temporarily remember the result of a task, when necessary, though. These partial results are stored in 'boxes' in the computer memory. Three points become obvious from the fact that the computer uses 'fingers' for its operations.

1. Only whole numbers can be represented directly by the CPU. As you may know, these whole numbers are called **INTEGERS**. The reason for this is that the CPU cannot work in 'half fingers'!
2. The size of the numbers that can be represented on the CPU fingers is limited by the number of fingers that the CPU has.

3. The limitation of point (2) above can be partially overcome by allowing the CPU to use its 'toes' as well as its fingers for its counting operations.

However, whereas we're stuck with two hands and two feet, the CPU has several more hands and feet and each 'hand' has 8 fingers and each 'foot' has 16 toes! In addition, by a clever coding system the CPU can count up to 255 on each of its hands and up to 65535 on each of its feet! This is clearly much more efficient than our usual way of counting on our fingers, and we'll take a look at how this coding system works in Chapter 2.

So, how might the CPU do a simple job, like add together 2 small numbers. Let's see how the sum '3+4' would be dealt with. First of all, let's give one of the CPU hands a name; we'll call it 'A'. Also, we'll let the CPU use some of the 'boxes' in the computer memory. The steps that the CPU might take to add the two numbers could be:

```
LD    A,3
LD    (BOX#1),A
LD    A,4
ADD   A,(BOX#1)
LD    (BOX#2),A
```

'LD' is a mnemonic for a machine code instruction called LOAD; this simply tells the CPU, in the first instruction, to count '3' on to its 'A' hand. In the second instruction, we tell the CPU to count the contents of 'A', which is 3, onto the 'fingers' of the memory box BOX#1. 4 is then counted onto A, and the value on A is added to the value in BOX#1. Finally, the result is stored in BOX#2.

The second instruction introduces us to a quite important concept of machine code programming. The brackets around the word 'BOX#1' indicate that we're interested in the *contents* of this box. In the first instruction, we were interested in the actual number '3', so we didn't need the brackets. The use of brackets in this way to refer to the contents of a memory 'box' is a bit like the idea of BASIC variables. If it helps at all, think of

the second instruction as LET BOX#1=A, where A can be thought of as another 'variable', in this case holding 3. The ADD instruction, by the way, leaves the result of the addition on the fingers of 'A'; this is very common in machine code programming. The results of CPU operations are always left on the fingers of various CPU hands. As you can see, it's a lot more long winded than the BASIC equivalent of LET A=3+4!

The Stack

Despite the apparent excess of hands and feet possessed by the CPU, it's often desirable for the CPU to be able to store results of CPU operations elsewhere. As we've seen above, the CPU can use memory 'boxes', but the CPU can also use a special form of storage called a **STACK**. Years ago, before in-trays, out-trays and computer filing systems were invented many desks would be occupied by a piece of wood with a metal spike stuck in it. Pieces of paper entering the office were stuck on this spike until they could be dealt with. The last piece of paper placed on the stack would thus be the most easily accessible of all the pieces of paper on the spike. The computer equivalent of the spike, the Stack, is useful to the CPU precisely because of this fact; if a piece of information has been stacked, the CPU knows where to find it. The CPU **PUSHES** information onto the stack, usually two hands worth of information at a time. A typical use is where the CPU has some information on a hand, but needs to use the hand for something else. PUSHing the hand on to the stack will save it, and once the hand is free again the CPU can **POP** the information back off of the stack onto the desired hand. Information can be stored from as many of the hands and feet of the CPU as you like, each store operation needing a separate PUSH. However, like the office spike, the only piece of information that is readily available is the last item PUSHed on. The office spike grew upwards in size whenever a piece of paper was pushed on to it. The stack also grows, but in the computer the stack is arranged so that it's upside down. Thus, the stack grows downwards in the computer as more information is PUSHed on to it.

What the CPU is Capable of

As we mentioned earlier in the Chapter, the CPU is really only capable of performing simple tasks. Due to the limitations set by the number of its fingers and toes, it is limited to using numbers in the following ranges:

1. 8 fingered numbers in the range 0 to 255.
2. 16 fingered numbers in the range 0 to 65535.

I use the phrase 8 and 16 fingered numbers deliberately here; this is because the CPU can use two of its 'hands' as an extra 'foot' if it wants to, thus effectively giving us an extra 16 fingered foot to put larger numbers on. The basic operations that the CPU can perform on these numbers are as follows:

1. Counting on one hand.
2. Counting on two hands.
3. Addition and Subtraction on one hand.
4. Addition and Subtraction on two hands.
5. Various manipulations of one handed numbers, such as making a number negative.
6. Causing the CPU to jump from one place in a machine code program to another.
7. Causing the CPU to transfer numbers from one part of the computer to another.

Before we leave this introductory Chapter, let's look at two last topics; the concept of **ADDRESSES** and the hardware of the Spectrum 128/Plus 2.

Addresses

In normal usage, an address refers to where abouts a particular building can be found in a town or city full of them. In computing terms, an address refers to where in the computer a particular 'box', used for the storage of numbers, can be found. The number in this box could be a machine code instruction or a piece of data. Whatever it is, it will be stored in the box as an 8 fingered number; this is the largest value

number that one of the boxes can hold. In the Spectrum 128, some of the boxes are used to store the BASIC Interpreter program, others store our BASIC program and still others are used to store the information that makes up the screen image.

Spectrum 128 Hardware

Let's now take a detour and look at the hardware of the Spectrum 128/Plus 2. **Hardware** is the term applied to the various electronic components that make up the computer system. I once read that hardware was the bit you could kick; this is, in some ways, true, but don't take it literally! **Software** is the term given to the programs that are run on computers. Figure 1.2 indicates how the various components of the system are arranged. Let's now look at the function of each part of the system.

The Z80 CPU

There are 8 hands in the CPU, and they've all been given names. They're called **A,B,C,D,E,F,H** and **L**. In addition, there are two feet, called **IX** and **IY**. These hands and feet are often represented diagramatically as shown in Figure 1.1.

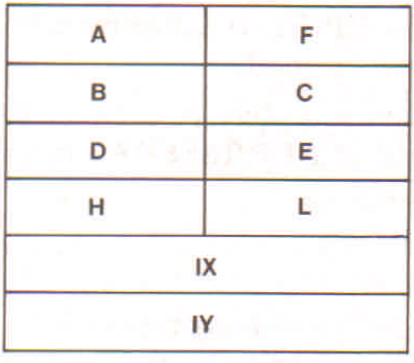


Figure 1.1 The Z80 Register Set.

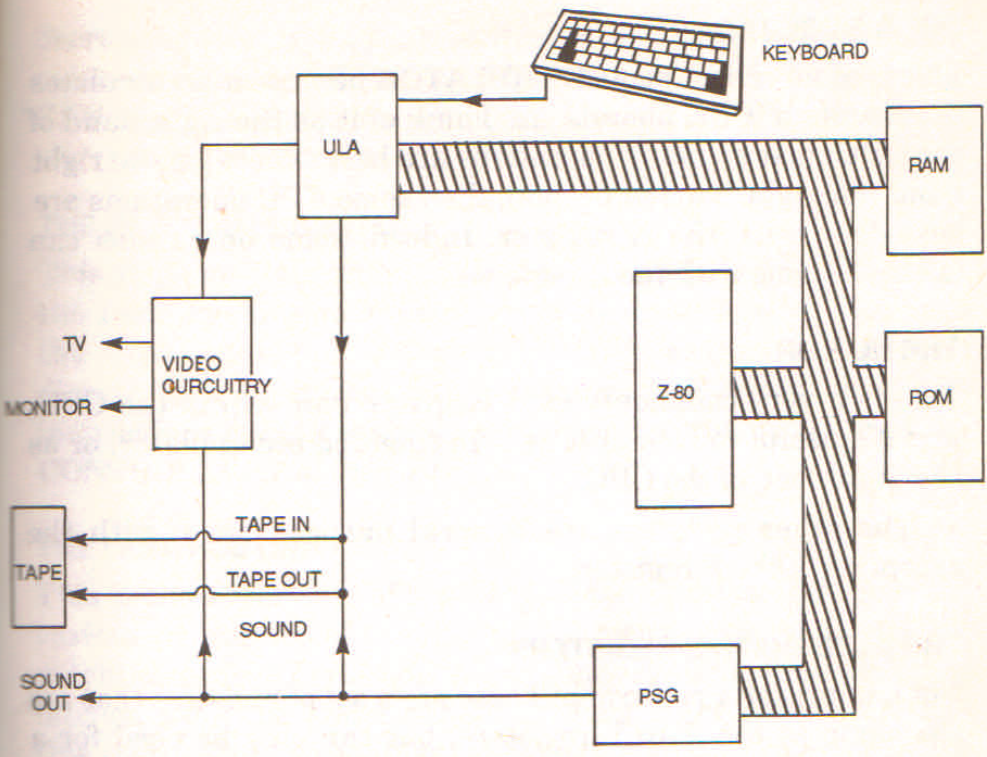


Figure 1.2 Spectrum 128 Hardware.

All of the hands except for 'F' can be used for counting on. Hand F has a special function, as each finger of this hand is used to indicate whether or not a particular event has happened within the CPU. We'll look at this hand in a later Chapter.

We can team up hands to form some new feet. The hands involved are B,C,D,E,H and L. The new feet formed are BC, DE and HL. Note that we can't just combine hands as we want; these are the only allowable combinations. A CE or BL pair is not possible. Each of these new 'feet' can hold a sixteen finger number, just like the IY or IX feet.

The hands and feet of the CPU are really called **CPU REGISTERS**. Thus the A hand is usually called the A Register. The feet produced by pairing up two hands, such as BC, DE or HL are called **REGISTER PAIRS**. The IX and IY feet are also given a special name; they are called **INDEX REGISTERS**. Don't worry about the Index Registers, as we'll be looking at these registers later.

THE A REGISTER

This is often called the **ACCUMULATOR** because it accumulates the results of CPU operations. Think of it as the right hand of the CPU; just as many operations are best done with the right hand (for right handed people), then some CPU operations are best done with the A register. Indeed, some operations can ONLY be done with the A register.

THE HL PAIR

This is a very commonly used Register Pair within the CPU, and it's useful to look at it as a 16 fingered accumulator, or as the 'right foot' of the CPU.

The other registers are 'general purpose' ones, with the exception of the F register.

THE ALTERNATIVE REGISTERS

Not a name for a pop group! These are a set of registers that are the same as the A to L registers, but can only be used for a limited range of tasks. They are called the **ALTERNATIVE REGISTER SET** and are called **A',B',C',D',E',F',H'** and **L'**. There are no alternative IX or IY registers. The only thing that we can do with these registers is to copy the contents of the main register set (Registers A to L) into them for safe keeping whilst we use the main registers for something else. When we do this, the contents of the Alternative Registers are copied into the main registers. When writing machine language programs on the Spectrum 128/Plus 2 it's a good idea not to use these Alternative Registers until you gain some experience. This is because the BASIC Interpreter makes frequent use of the Alternative Registers, and would get rather confused if you altered their contents and didn't change them back!

THE STACK POINTER

This is a special Register Pair that we don't directly use very much. It's used by the CPU to indicate the address of the position in memory of the last entry on the Stack. As the Stack grows downwards into memory, the number in this register

decreases with each PUSH and increases with each POP. The contents of the Stack Pointer (SP) is thus altered whenever the CPU uses the Stack. This is done automatically, so we needn't worry too much about it.

THE PROGRAM COUNTER

This Register Pair tells the CPU where in memory it can find the next machine code instruction so that the CPU can fetch the instruction and act on it. The Program Counter isn't directly altered by our programming. The collection of the instructions is dealt with by another part of the CPU called the **CONTROL UNIT** of the CPU.

THE CONTROL UNIT

This is the supervisor of the CPU. It coordinates and times the various operations of the Z80 and is responsible for fetching a machine code instruction from memory. The instruction is fetched from the address held in the Program Counter, and this instruction is then passed over to a special register in the CPU called the **Instruction Register.**

THE INSTRUCTION REGISTER

This register holds an 8 finger number representing an instruction while the CPU works out what the instruction is. One the task to be done is known, the Control Unit acts upon it.

THE ARITHMETIC AND LOGIC UNIT

This is often called the **ALU**, and is the 'pocket calculator' of the CPU, being responsible for all the arithmetic operations that the Z80 can perform. It's very simple in its function, being able to add and subtract but not being able to do multiplication or division. It can also perform what are called **LOGICAL** operations on the contents of CPU registers, such as comparing the contents of two registers, and raising (setting to '1') or lowering (setting to '0') fingers within a hand. As a byproduct of the activities of the ALU, the fingers in the F register are affected.

Despite the fact that the Z80 is a very smart device, it would be useless without the other chips in the 128. Let's now take a quick 'Cook"s Tour' around them to see what they do.

Memory

Because of its 16 fingered Program Counter, the Z80 can gain access to 65536 different 'boxes' in memory. However, as we'll soon see, by a clever feat of engineering the 128 uses some of these locations 'twice', thus allowing us to gain access to more memory. This is used by the 'Silicon Disc' facillity of the 128, but we can also use it for storage if we like. However, we won't go into it in too much detail; this is, after all, a beginner's book.

The Spectrum 128 has two types of memory, called **Read Only Memory** and **Random Access Memory**. Let's now look at what these names mean.

READ ONLY MEMORY

This type of memory is used in the Spectrum 128 to hold the BASIC Interpreter. ROM, as this type of memory is also known, has a useful ability; it retains its contents when you turn the power off. Thus, the BASIC Interpreter is always in your machine. However, we cannot alter the contents of ROM, except by putting new memory chips into the machine! We can still copy numbers held in this type of memory into CPU registers, and the CPU can run programs made up of instructions held in ROM. However, we cannot write any new information from registers into ROM. Hence the name, Read Only memory.

Well, ROM is alright for programs or information that we don't want to alter. But what about the programs we write? We can type them in and the computer remembers them. But, as you know, when we turn off the power to the 128 our BASIC programs disappear from the computer memory, hence the need for SAVEing them to tape or Microdrive. There must, therefore, be another type of memory apart from ROM.

RANDOM ACCESS MEMORY

For this type of memory, I prefer to use the unofficial but much more descriptive name of Read and Alter Memory. This describes exactly what RAM is; a type of memory which we can alter at will. We can read number from it, then alter it, as often as we want. Thus for situations where we frequently need to change the contents of memory, such as BASIC or machine code programming, we use RAM for our memory.

We've already mentioned the big drawback with RAM; turn off the power, and the contents are lost forever. Just as with ROM, a 'box' in RAM can store 8 finger numbers – between 0 and 255. Later in the book we'll see how we can use two of these locations to hold a 16 finger number, but for now let's just stick to 8 finger numbers in memory locations.

Before leaving memory for a while (we'll return to it in Chapter 3), how does the CPU tell the memory, whether ROM or RAM, which 'box', or LOCATION, it wants to use? Well, the CPU is connected to memory by the 8 pins that we mentioned at the beginning of this Chapter, called the **DATA** pins of the CPU because they carry information, or data, and 16 other pins called the **ADDRESS** pins. These tell the rest of the computer what value is in the Program Counter at any time. When the address of a memory location is put into the Program Counter, therefore, the memory knows about it, and the correct memory location will 'know' that the CPU wants to access it and so will make itself available to the CPU. Other pins of the CPU tell the memory whether a read or write operation is needed. Remember that reading takes information FROM memory TO the CPU registers and a write takes information from the CPU registers TO the memory. The collection of Address pins is called, in technical jargon, the **Address Bus,** and the collection of Data pins is called the **Data Bus**. The 'bus', in this case, carries numbers around the computer circuit rather than people around a town or city! Remember that although the CPU can only access, or **ADDRESS**, 65536 memory locations at once the Spectrum 128/Plus 2 has about twice this many memory locations. Exactly how the 128 gets around this problem will be discussed later.

The Programmable Sound Generator

This chip, also called the PSG, is responsible for the sounds produced with the PLAY command. BEEP uses a different technique to produce sounds. In addition, the PSG is responsible for the MIDI interface, which allows the Spectrum 128 to control suitable electronic musical instruments such as synthesisers, and the RS232 Serial Interface, which allows the 128 to communicate with suitable printers, other computers or telecommunications equipment. We'll look at the PSG in considerable detail in Chapter 12.

Video Circuitry

This collection of electronic components is responsible for producing the image that you see on the Television or monitor screen. The Plus 2 can drive a standard television, an RGB monitor for colour displays or a green screen monitor for cases where a steady display is more useful, such as might be required for long periods of programming. There's nothing here that we can program.

Uncommitted Logic Array

This is a 'black box' amongst the chips in the 128. Suffice to say that it helps the CPU out with many of the housekeeping activities, such as screen display and keyboard reading. It occasionally causes us some problems, especially if we're putting machine code programs in the lower half of memory. However, these problems are quite rare and I'll mention them when we come to them. The ULA is probably the second most important chip in the computer after the Z80.

All these devices are under CPU control to some extent, with the exception of the Video Circuitry. If we want to alter their behaviour, we have to program the CPU accordingly. However, this isn't as difficult as it sounds, and we'll see later in the book how we can use machine code to get the best out of the 128 hardware. Now that we've discussed the cast list of the Spectrum 128/Plus 2, let's go on to look at something that is rather fundamental to computers; 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 its 8 fingered hands. How can this be, when we can only count to 10 on our fingers? Well, the answer is that the way in which we count on our fingers is rather inefficient, and the computer is simply more efficient in its counting than we are.

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's no reason why this should be. We could let each finger represent a different value. For example, a raised first finger could indicate a value of 1, a raised second finger 2 and so on. In this scheme, the number 3 could be represented on just two fingers by raising the first and second finger (1+2). When either finger is lowered, it has a value of zero. Thus to represent the number 2, we'd just raise the second finger and keep the first finger lowered. This method is clearly more efficient than our usual means of counting on fingers, as the usual method of counting on our fingers would need 3 fingers to represent the number 3, and this new method needs only 2 fingers. The counting method used by the CPU is based upon this idea, and appreciates the below facts:

1. Whether a finger is lowered or raised is important to the overall number being represented on the fingers.
2. The position of a finger within the hand is important to the value represented on that finger, which in turn is important to the number represented on the hand as a whole.

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

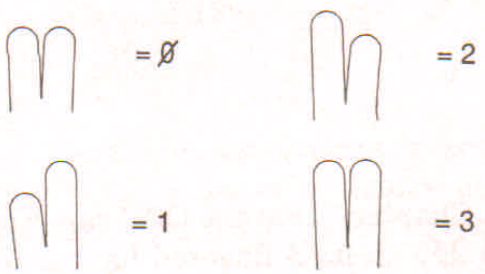


Figure 2.1

We might represent a raised finger by the digit '1' and a lowered finger by the digit '0'. This is much easier than drawing pictures of hands all over the place. Thus we could rewrite the above as:

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

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

Such a method of representing numbers in which there are only two different states (raised or lowered fingers, 1 or 0) is called a **BINARY** method of representing numbers. If we add another finger to the two that we've considered so far, then we have 8 different combinations of three fingers, thus allowing us

to represent 8 different numbers. If you don't believe this then try it with your own fingers. We can represent the numbers 0 to 7 on these three fingers. Let's use our 0 and 1 notation to represent lowered and raised fingers.

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

The addition of a fourth finger allows us to represent numbers between 0 and 15. In computing circles, the numbers 10 to 15 are often represented by the letters A to F rather than the two digit numbers 10 to 15. Thus,

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

This method of representing numbers is called **HEXADECIMAL NOTATION.** In this way of representing numbers the number 0 to 15 are represented as 0,1,2,3,4,5,6, 7,8,9,A,B,C,D,E and F. 16 is represented as 10 in hexadecimal, 17 as 11 and so on. A four finger number can be represented by a single hexadecimal digit, and an 8 finger number can be represented by 2 hexadecimal numbers. We indicate that a number is in hexadecimal by prefixing it with '&' or '#' or following it with 'H'. Although the Spectrum doesn't understand hexadecimal numbers directly, there are advantages to be had in using them. Obviously, hexadecimal numbers can be converted in to decimal, and vice-versa. We'll soon see how we can do this. The advtantages offered by using hexadecimal notation are:

1. We can easily convert hexadecimal numbers into binary numbers and so we can see which fingers within a hand are raised or lowered.
2. By the number of hexadecimal digits used to represent the number, we can tell if the number will fit into one or two hands; a one handed number has two digits and a two handed number has four digits.

How can we work out what value a number has if we know which of the fingers are raised or lowered? Well, look at Figure 2.2.

Figure 2.2

We've got 4 fingers raised here, and we've assigned each finger a value. When any finger is raised it has the value given, and if it's lowered it has the value 0. Thus with all 4 fingers raised, as above, 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 was 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:

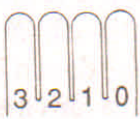


Figure 2.3

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 up to 15. You should be able to see what to do to enable us to represent larger numbers; just add more fingers. For example, the number 16 is represented on an 8 fingered hand in the below fashion, with finger number 4 raised.

Figure 2.4

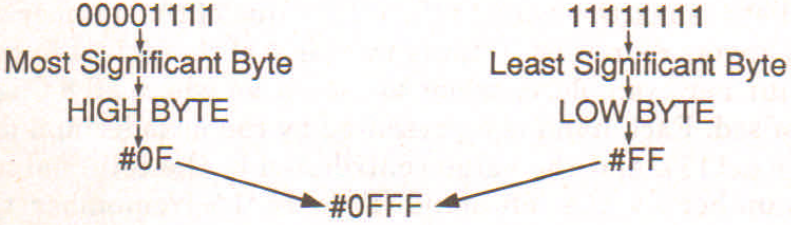
This can be written in hexadecimal, as we've already seen, as #10. We arrive at this by splitting the 8 finger number into two 4 finger numbers, and we then give each 4 finger number a separate hexadecimal digit. Thus in this case, the right 4 fingers are lowered, thus we can represent them by a '0'. Of the left 4 fingers, the rightmost of them is raised, so we can represent these four fingers with a '1'. The significance of these 4 finger 'handlets' is not the same to the total value of the number; the left 4 fingers represent 16 times the value of the right 4 fingers. As a further example, consider the situation where all 8 fingers are raised. Each hand is represented by the hexadecimal digit 'F' (#F=1111), and the value contributed to the total value of the number by the left hand 'F' is 16*15 (remember that #F=15). The value contributed by the rightmost 'F' is just 15, so the total is 15*16+15, or 255. Thus, we can see how we can represent 255 on 8 fingers. An extension of this will allow us to represent 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's the same in computing. Each finger, or binary number, is thus called a digit, and there are 8 such digits in our 8 fingered numbers. There is a special name for a Binary Digit – we call it

a **BIT**. This is simply a shortened version of BInary digiT. We can thus call the 8 finger numbers 8 bit numbers. These collections of 8 bits are also given a special name; they're called **BYTES**. A byte is thus a number that can be represented on 8 fingers or a CPU hand. The 4 finger 'handlets' are called NIBBLES, a nibble, after all, being a small byte.

The terms bit, byte and nibble are all common in computing circles, and you'll come across them in this book and in other books as well. Just as we numbered our fingers, we give the bits within the byte a number. We number the bits from right to left, 0 to 7. Bit 0 is given a special name, the **LEAST SIGNIFICANT BIT**, or LSB. This is because the value contributed to the byte by this bit, 1, contributes the least to the value represented by the byte. For a similar reason, bit 7 is called the **MOST SIGNIFICANT BIT**, or **MSB** of the byte. In the same way, we also label the two bytes that make up a 16 bit number of the type that can fit into a register pair.



Here the high byte has a significance to the total value of the 16 bit number of 256 times the low byte. Thus the total value of a 16 bit number is given by:

TOTAL = 256*HIGH + LOW

where low is the value of the low byte and high the value of the high byte. In the same way, a 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 name of the High Register is written first. Thus in the BC register pair, B is the high register and C is the low register. This fact isn't too difficult to remember if you think about the HL register pair.H holds the High byte and L the Low byte. This might be why the

name 'G' wasn't applied to a register; a GH register pair might be a little confusing.

Representation of Information

Human beings deal with information mainly as numbers and letters. Computers can only deal with numbers. Therefore, because the computer has to tell us the results of its operations, it's clear that the computer must have a way of representing other forms of information. There are two main types of information represented by numbers in a computer:

1. Machine language programs. These could be the BASIC Interpreter or a program written by the user.
2. Data for a machine language program. This can either be numeric, or might include some textual information. A BASIC program, for example, could be treated as data for the BASIC Interpreter program.

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

Program Representation

A machine language program, as we've already said, is a sequence of bytes that represent machine code instructions to the CPU of the computer. They are stored in the memory of the computer, and Z80 instructions can be between 1 and 4 bytes in length. For example, the HALT instruction is only one byte in length. Once the number representing the instruction, 118, has been read from memory and acted upon the CPU is able to go on and get the next instruction. However, with the more 'long winded' instructions the CPU cannot do anything until it has read the rest of the instruction from memory. The Control Unit of the CPU automatically knows how many bytes are needed to make up a given instruction. It should be obvious that a single byte instruction will probably be acted upon more quickly than one of these 'multi byte' instructions.

Data

In BASIC we are able to use various types of variable to hold information on which our programs are to work. These include whole numbers, such as 1,2 or 4000, Real Numbers, such as 1.23, 0.34 or 0.000001 or strings of characters, such as "Hello" or "Don't Panic". In machine code, we don't have this sort of versatility; the only numbers that the CPU can handle directly are those in the range 0 to 65535. To represent Real Numbers, or numbers outside the range above, we have to program the CPU to do so. Character strings are available with a little programming, and letters are stored as numbers in the memory. Programming the CPU to handle real numbers is quite a job.

Integers

Integers, or whole numbers, are easily dealt with by the CPU provided that they're in the range 0 to 65535.

Signed Integers

These are integers which can be negative. Thus -1 is a signed integer. Like all numbers in the computer, the CPU sees the numbers as collections of 1s and 0s. We need a means of telling the CPU whether a number is negative or positive, just as we use '+' or '-' in normal arithmetic. The most commonly used method of representing the sign of a number is where we use the most significant bit of the number to represent the sign. If such a number is negative, then the MSB is set to 1 and if it's positive then the MSB is set to 0. Thus for an 8 bit number, we've got bits 0 to 6 in which to represent the value of the number and bit 7 to represent the sign of the number. Bit 7 is often thus called the **SIGN BIT** of an 8 bit number. The resultant 7 bits don't hold values in the range 0 to 255 any more. Instead, half the numbers represented will be below 0 (sign bit set to 1) and half above 0 (sign bit set to 0). The new range of numbers represented will be -128 to +127. This gives us a problem; for any 8 bit number, how do we tell whether an 8 bit number is a negative number or a large positive number?

The answer is that we don't. It depends purely upon what interpretation the programmer puts on the numbers at any time. The machine code instructions don't care what interpretation we put on them, but the interpretation we put on the result of such operations depends upon the representation used. Producing a negative number is not, however, just a matter of setting bit 7 to 1. We must also alter the way in which the lower 7 bits of the byte hold the value of the number. The fundamental thing to remember about a negative number is that if you add it to the corresponding positive number you get zero. E.g.

(+1) + (-1) = 0

Thus the binary representation of -1 must be such that when it's 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 0 (1+1=0 carry 1 in binary) but what about the other bits?

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

Well, this isn't the correct result. We really need to take the carry that was generated by the addition of 1+1 and use it to somehow set all the other bits in the result to zero. This requires that all the bits in the binary representation of -1 be set to 1.

```
  00000001
+ 11111111
  --------
  00000000
```

Well, this is the correct answer for -1, but we must really try to work out some general rule that allows up to work out the representation of a negative number. Well, let's start with -1. The representation of -1 above was obtained in two stages:

1. Replace all the 0s in the the positive number with 1s and all the 1s in the postive number with 0s. The number is thus:

   00000001 goes to 11111110

This process is called COMPLEMENTING a number. the complement of 1 is 0 and the complement of 0 is 1.

2. Add 1 to the result of complementing.

   11111110 + 00000001 = 11111111

The only way to see if this method gives us a true binary representation of a negative number is to try applying it to other numbers. Let's see if it works on the number -2.

1. Complement the number +2. This gives us:

   00000010 goes to 11111101

2. Now add 1 to this:

   11111101 + 00000001 = 11111110

This should be the correct representation of -2. To see if we're right, let's add this representation to +2:

```
  00000010
+ 11111110
----------
  00000000
```

Yes, that's correct, as we get the result zero. This method of representing a negative number in binary is called the **TWOS COMPLEMENT** representation. It's the most common form of representation of negative numbers. If you apply the methods to a negative representation, then you'll get the positive number back. This isn't really surprising once we remember that two minuses make a plus.

So far, we've only applied this to 8 bit numbers. However, it works just as well with 16 bit numbers When we use twos complement notation to represent negative 16 bit numbers, the MSB of the high byte is the sign bit rather than bit 7. The complementing operation is just the same. 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.

Characters and Strings

We rely on textual information a great deal, and so it's important that we should be able to represent letters in the computer as well as numbers. Well, all that we do is allow each character, which is a letter, number or anything else that we can display on the screen of the 128 with a PRINT command, to be represented in the computer by a number. Again, it's a case of what the programmer wants a particular number to be. The limit of 255 different characters available to the Spectrum 128 is due to the fact that the characters are stored as single byte numbers in memory. To ensure that we all maintain our sanity, the numbers used to represent particular characters have been standardised, so that in different computers we use the same numbers to represent the same characters. The most popular code used is called **ASCII**, which stands for 'American Standard Code for Information Interchange' – quite a mouthful. As an example, the code for 'A' is 65 and that for 'a' is 97. To find out the code for a particular character, just type in PRINT CODE a$, where a$ holds the character of interest.

Summing Up

You can thus see that to a large extent the meaning of a particular number in a memory location in the computer depends upon what the programmer wants it to represent. A byte can be:

1. A machine language instruction.
2. A number in the range 0 to 255.
3. A number in the range -128 to 127.

4. Part of a 16 bit number or a multi byte instruction.

5. A character code.

It is thus important for the programmer to keep an eye on what he uses different parts of the computer memory for. For example, the CPU might try and treat the bytes making up the message 'Hello There!' as a program! This would result in a very confused CPU.

Well, having seen what machine code instructions are, and how computers count, it's about time we looked at the memory of the Spectrum 128 in more detail and find out exactly how we can enter machine code programs into memory.

Chapter 3
Machine Code and BASIC

In this Chapter we'll look at how we can enter machine code programs into the computer, and see where in the machine we can store the programs that we write.

When we turn the Spectrum on, the BASIC interpreter begins running and expects us to enter BASIC commands or programs. Therefore, whenever we type in machine code, somewhere along the way is going to be at least 1 BASIC command, even if it's just the instruction to tell BASIC to run our machine code program rather than the BASIC Interpreter.

Homes for Bytes

The machine code programs that we'll deal with are all made up of sequences of bytes representing machine language instructions. Therefore, one of the first things to do is to find a place in the computer memory where these bytes can be stored without them being altered in any way.

For programs that we are writing, it's clear that this has got to be in RAM; remember that we cannot affect the values held in ROM. Although there's a good bit of RAM in the Spectrum, some of it is used by the Spectrum BASIC Interpreter and are unsuitable for our machine code programs. Such areas are

those used for the storage of your BASIC programs or variables, the screen image, or areas used by the Z80 as 'scrap paper' while it executes the BASIC Interpreter program. Any machine code programs placed in these areas are thus prone to being overwritten by the BASIC program as you add more lines or create more variables, by the images we put on the screen or by the Z80 as it executes the OS programs. These areas of memory that are used by the CPU to perform its normal tasks are collectively called WORKSPACE. If we alter workspace locations, there is a good chance that we could alter some of the information that is required by the CPU to execute the programs properly. The Z80 would then, not surprisingly, lose track of what it was doing and 'crash'. This is as unpleasant as it sounds, the CPU going out of control. The results of a crash are usually the computer resetting itself, with the subsequent loss of anything in the machine. Alternatively, it might just sit there, keyboard not operating, screen not changing, until you put it out of its misery by pulling the power plug. This, by the way, is often called the 'Sinclair RESET'!

The moral of all this is that when we're playing with machine code, it's a good idea to save programs to tape or microdrive before running them. That way, if we crash we can get back the material that was in the machine and correct the problem without us having to retype the whole lot in.

When you start programming in machine code, you will have a few of these crashes as you get used to programming. This is because, compared to BASIC, machine code is very unforgiving with mistakes. You can't enter a 'dud' line of BASIC into the Spectrum, but machine code offers you no such 'Syntax Checks'. In addition, whereas errors such as 'No such variable' in BASIC are indicated to the programmer who can then correct the problem, no such niceties exist in machine code. If you are lucky, then the unexpected will happen when you make a mistake in your programming; if you're unlucky, then a crash can be the result.

Anyway, let's get back to the problem of finding somewhere to put our machine code programs. The parts of memory that

are used by the Spectrum BASIC Interpreter are shown in Figures 3.1 and 3.2. These **MEMORY MAPS** simply show which bytes in the computer memory are used for what purpose. As you can see, some addresses in the memory, such as those between 0 and 16384, are occupied by two different pieces of memory! This is how we can fit the computing 'quart' of 128k of memory, into the 'pint pot' of the Z80, which should only be able to address 65536 different locations in memory (addresses 0 to 65535). This sharing of address space is called PAGING, because it's similar to the pages in a book; the pages occupy the same space on the desk but you can look at many different pages simply by turning them over.

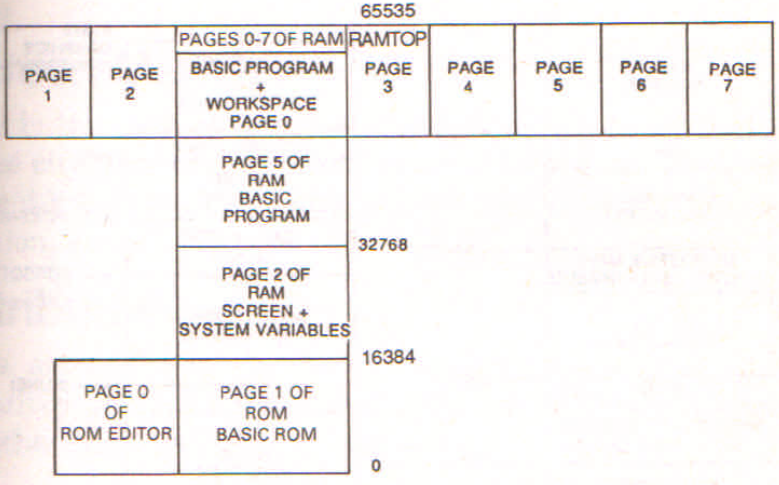


Figure 3.1 Simplified Memory Map.

We'll take a closer look at this in a later Chapter. All that we need to know at the moment is that when we are running a BASIC program, or in 48k mode, the BASIC Interpreter ROM is '**paged in**' and is used by the CPU. When we are editing a program in 128 mode then the Editor ROM is in use. However, when a BASIC program is being run in 128 mode the BASIC ROM is active again.

RAM is also paged in the Spectrum 128; there are 8 'pages' of RAM, all of which contain 16384 bytes of memory. They are all paged in to the top 16384 bytes of the memory map, as can be seen from Figure 3.1. Just to complicate matters a little, two of these pages, Page 2 and Page 5, are also 'paged in' to other

places in the memory map, as can be seen. They are used for the screen and BASIC program storage. Again, we'll look at RAM paging in a later Chapter. For now, just be aware that normally we are using pages 0,2 and 5 of RAM, page 0 occupying the top 16384 bytes of memory and the others being used where indicated in Fig. 3.1. However, when we're using the BASIC Editor in 128 mode, page 7 of RAM is used instead of Page 0. We also use the paged RAM when we use the SAVE !, LOAD ! etc. commands to use the 'silicon disc' facility of the computer.

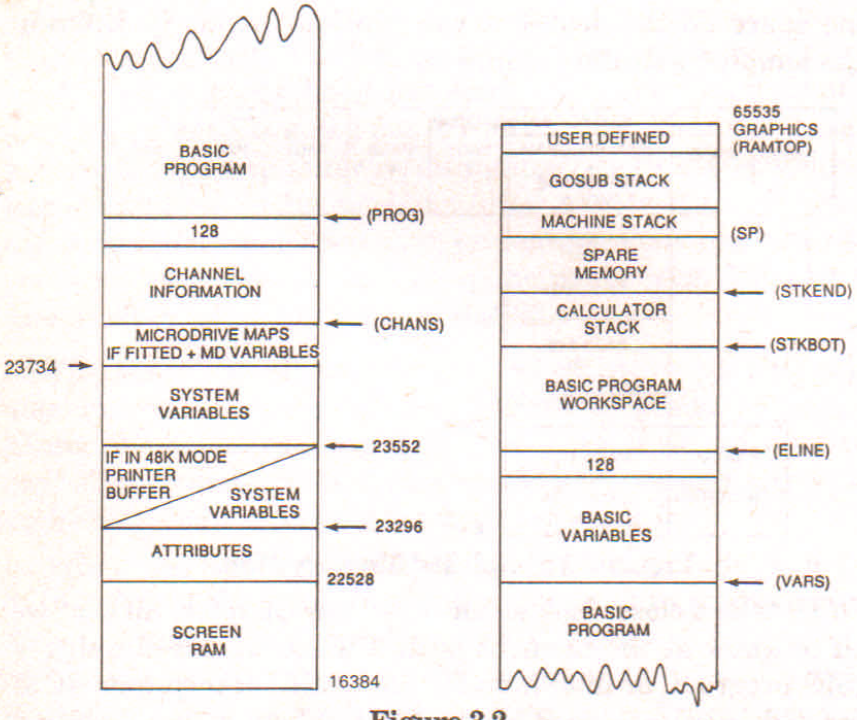


Figure 3.2

Let's examine the maps in a little more detail. The addresses are in decimal, or are indicated as **LABELS**. These are such things as (PROG) or (VARS). These names refer to places in the memory that hold the actual address of the start of the BASIC program (PROG) or of the start of the area of memory that is used to store the BASIC variables in (VARS). The reason why these values need to be stored in the workspace is simply that the exact position in memory where, for example, the variable

storage starts depends on the length of the program, amongst other things. So, the CPU keeps an eye on how things are going and updates the stored addresses when it needs to. Thus, if we add another line of BASIC to a program, the value held in the the memory locations used to store VARS will be changed. All these 'labels' are stored in the area of memory called 'System Variables' in Figure 3.2. There are many of these, some of which we can make good use of in our programs. The more useful of these we'll look at later in the book. For now, just treat the system variables as 16 bit numbers that represent a particular address in the computer memory map, or represent certain facts about the BASIC Interpreter to the CPU.

Screen Memory

This is the area of memory used to hold the bytes that tell the video circuitry of the computer what to put on the screen. It's over 6000 bytes long, and is arranged in a slightly peculiar fashion, as we'll later see.

Attributes

This area of memory holds the information for the video circuitry about the colour of different things on the screen, whether they are flashing, and so on.

Printer Buffer

In 48k mode, this is used by the Operating System to allow information to be sent to the ZX Printer. A line at a time of text is formed here and it's then sent to the printer. In 128 mode this area is used to provide space for the extra system variables needed to run the keypad, handle the serial interface and so on. In 48k mode, if you're not using a printer, therefore, you can use the printer buffer space for short machine code programs.

System Variables

This area of memory holds the System Variables and is totally OFF LIMITS for the storage of machine code programs!

Microdrive Maps

This area of memory doesn't always exist, depending upon whether you've got Interface 1 fitted and microdrives in use. When it does exist, it's used to give the Operating System an idea of the way in which different parts of the microdrive tape have been used.

Channel Information

This area of memory holds the information necessary for the BASIC Interpreter to use the screen, keyboard, and, where possible, the ZX printer. This finishes with a byte holding the value 128.

BASIC Program

This is stored from the address in (PROG) onwards. After the program we have the area of memory devoted to the BASIC variables, and the start of the variable area of memory is held in System Variable VARS. A useful application of these two system variables is to get the length of the program. This is done by subtracting the value of PROG from that or VARS. PROG is held in bytes 25635 and 23636, low byte first. VARS is stored in addresses 23627 and 23628, again low byte first. The end of the variables area is marked by a byte containing 128, and this is followed by the BASIC program workspace.

The Calculator Stack

The Spectrum ROM contains a lengthy program called the **Floating Point Calculator.** This is used when the CPU needs to do complex arithmetic, such as multiplication, doing sines and cosines, etc. This 'stack' is used for temporary storage of results calculated by the Floating Point Calculator.

The Machine Stack

This is the stack pointed to by the Stack Pointer.

The GOSUB Stack

This stack is used to store the return addresses for BASIC GOSUBs. The information held on this stack tells the BASIC interpreter where to go when a RETURN instruction is encountered in a BASIC program.

RAMTOP

This is the last byte of memory available to BASIC. Anything above this isn't affected by the BASIC program. We can 'move' this address around in memory, though we must use care. Setting RAMTOP too low, for example, will lead to a 'memory full' error, even though there will be many k of memory above RAMTOP. Usually, the only thing above RAMTOP are the definitions of the User Defined Graphics characters, accessed by USR("a") and so on. However, if we move RAMTOP down, we have some 'spare space' between RAMTOP and the definition of the User Defined Graphics. This space is protected from incursions from BASIC, and so would make a great place in which to put our machine code. That is in fact what we do. This is the 'official' method of generating space in the Plus2/128 for machine code. To set RAMTOP to a particular value, we issue a special form of the **CLEAR** command:

```
CLEAR    nn
```

where nn is a 16 bit address which specifies the last available byte to BASIC. Thus the command:

```
CLEAR    39999
```

will set RAMTOP to address 39999 and the first protected byte is one after this; i.e. 40000. Thus all the memory from 40000 to the start of the UDG definitions would be available for machine code. This command should be one of the first you issue when programming, and it stays in force until you issue another CLEAR nn command, or until you reset the machine or change from 48k to 128k mode or vice versa.

While we're talking about memory, let's introduce a little more jargon that you may encounter. It's a problem with

memory that we tend to end up talking about long strings of numbers when describing the quantity of memory possessed by computers. For example, the Z80 can address 65536 different locations of memory. To cut the numbers down, we say we've got 64k of memory, where a 'k' is 1024 bytes of memory. So, 2048 bytes are 2k, and 1/2k is 512 bytes of memory. The reason why a computer 'k' is 1024 and not 1000 is that 1024 is a whole number power of 2; 1000 isn't.

So, now we've seen how we can store our bytes in memory, safely above RAMTOP, can we put them there in the first place? Also, can we look at the contents of memory locations from BASIC? The answer to these questions is 'yes' in both cases.

POKE and PEEK

When we want to put a byte into a particular memory location we use the rather descriptively named POKE command. This is exactly what it does; it POKEs, or puts, a value into a particular memory location. It's used in the following fashion:

```
POKE address,value
```

The address must be in the range 0 to 65535, and the value must be in the range -255 to +255. If this isn't so, then an error will be reported. 'address' is the address of the memory location into which we want to put the 'value'. If the address isn't a RAM location, then the value will obviously NOT be inserted; you can't alter ROM! Take care with POKE, though; don't alter the System Variables unless you're very sure you know what you're doing. Thus the command:

```
POKE 40000,201
```

will put the value 201 into address 40000. So, we can use POKE to put the bytes that represent our machine code program into memory. What about looking at the bytes once they're there? This is done by using the PEEK command, which allows us to look into a memory location, whether it's ROM or RAM. As

PEEK is a function, we can PRINT its value out or assign its value to a variable.

```
PRINT PEEK(address)
```

will print to the screen the value held in the byte at address 'address'. Similarly,

```
LET a=PEEK(address)
```

will put the value in byte 'address' into variable 'a'.

Thus if we're in BASIC these two commnds will form the very heart of our attempts to input machine language routines in to the Spectrum memory.

USR(address)

Of course, once we've got the bytes into memory, we really want to be able to run the program that they represent. This is done by using another function called USR(address), which stands for **USer Routine**. When we use it, by, for example,

```
PRINT USR(address)
RANDOMIZE USR(address)
```

then the machine code routine represented by the bytes starting at address 'address' will be executed by the Z80. USR tells the CPU to leave the BASIC Interpreter and execute another program. When we are in a program called like this, we're 'on our own'; there is no error trapping and no break key. Thus:

```
PRINT USR(0)
```

will run the machine code routine that starts at address 0. This will reset the computer.

What about swapping information between BASIC and machine code? Well, there's no immediately obvious method of transferring information from BASIC variables to machine code routines, but on returning from a USR routine the BC register contents are passed back to BASIC. Thus in a statement such as PRINT USR(40000), the value printed to the screen would be the value that was in the 16 bit BC register

pair when the machine code routine finished. Thus it is easy to pass back some information to BASIC in the BC register pair. As to other methods of making BASIC variables available to machine code, or passing information back to BASIC programs in the variables, see Chapter 13.

The simplest method of passing numbers to and from BASIC, though, is to use POKE and PEEK in the following way. POKE is used to load bytes of memory with values to be accessed by a machine code program. The machine code program can then load registers with these values, and use them in the program. The result can then be stored in the memory locations for PEEK to recover after the routine has been executed. Alternatively, we can use the BC register pair to return values to BASIC.

Saving Bytes on Tape

You will occasionally want to store the bytes that represent a machine code program on tape; the easiest way to do this is to use the SAVE "filename" CODE address, length command from BASIC. 'address' is the start address of the area of memory to be saved and 'length' is the number of bytes to be saved. For example:

```
SAVE "fred" CODE 40000,100
```

will save the area of memory between 40000 and 40099. (100 bytes, including the byte at address 40000).

Loading Bytes from Tape

We use the LOAD "filename" CODE address, length command from BASIC to do this. 'address' is the address to which we want the bytes to be saved, and 'length' is the number of bytes that are to be loaded in from tape. The 'length' parameter isn't compulsory, but is useful if you have a number of files with the same name but different lengths on the same tape (very bad practice!!!).

Entering Bytes

There are three main methods by which we can enter bytes representing machine code programs in to the program:

1. Use an assembler program, which will allow you to type in the machine code programs in their mnemonic form.
2. Use a MONITOR program, which is a program in either BASIC or machine code that allows you to enter the bytes into memory in decimal or hexadecimal, and allows easy listing and editing of bytes in memory. Complex monitors also allow 'single stepping' of your machine code programs; here, the computer stops after executing each instruction in the program and displays the contents of the registers after that instruction.
3. Put the bytes into a DATA statement and then write a short BASIC program to POKE the bytes into memory. This is quite useful for the routines in this book, as the bytes can be entered, the BASIC program saved on tape and then the program run to enter the bytes into memory. USR can then be used to test the program.

The program to deal with (3) above can be quite simple:

```
10  CLEAR  39999
20  LET address=40000
30  READ byte
40  IF byte=999 THEN STOP
50  POKE address,byte
60  LET address=address+1
70  GOTO 30
80  DATA data representing the program.
```

The bytes are entered into the DATA statement just like any other data, each number being typed in followed by a ','. After the last byte of the machine code program type in the value '999' as the last item in the data statement to tell the program where the end of the machine code is.

Well, we're now ready to look at some instructions that the Z80 can understand. Before we do so, though, a look at an

instruction that's probably the most useful you'll use; it allows you to get back to BASIC from your machine code program. The instruction is called:

RET

which is short for RETURN. This is the equivalent of the BASIC RETURN command, which we use to finish subroutines off. Whereas forgetting a RETURN in BASIC will usually generate an error message, forgetting a RET in machine code will often result in a crash!

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 can't do much programming of the CPU until we can get some numbers into the CPU registers, and transfer these numbers between the CPU registers and the memory of the computer. So, in this chapter we'll look at the instructions that are available to us to allow us to move 8 bit numbers between different CPU registers and the memory of the computer. We'll look at how to deal with 16 bit numbers in a later chapter.

There's a little jargon to deal with here. When we are transferring data from one register to another register, or between memory and registers, the register which originally held the data is called the **SOURCE REGISTER**. That to which the data is going is called the **DESTINATION REGISTER**. Similar terms are used when we're transferring data between registers and memory. It's common to say that we LOAD a register from memory, or LOAD memory from a register. Indeed, the mnemonic that is used for these data transfer instructions, LD, is short for LoaD.

At this stage, it's important to realise that on the whole we COPY data from one register to another; after any of these operations, the destination AND the source registers hold the contents of the source register before the operation.

The designers of the CPU gave us a variety of different ways in which we can use the CPU. This isn't meant to confuse machine code programmers, but is intended to provide us with methods of solving problems that are both fast and convenient, as we'll later see. The ways in which the CPU is able to transfer data are known as the **ADDRESSING MODES** of the CPU, and virtually all the instructions in the Z80 instruction set can operate in at least one of these addressing modes.

I'll look at the addressing modes with respect to the LD instructions, although they're applicable to other instructions, as we'll later see. From now on we'll write our machine code programs down in the form of mnemonics. If you're using an assembler, then you'll be able to type in the programs as you see them. Otherwise, you'll have to hand assemble them, using the tables in the back of the book. To help you out, some of the first programs in this book will have the listings of machine code instructions ready assembled for you to enter into memory, as we saw in Chapter 3. So, let's get on with Addressing Modes.

Register Addressing

This is probably the simplest mode of all; it is used for operations involving two registers. With LD operations, therefore, this mode is used when data is being copied between two CPU registers. An example of this addressing mode is:

```
LD      A,B      load A from B
```

which simply copies the contents of the B register (source) into the A register (destination). The contents of the B register are totally unaffected by the transfer operation, but whatever was originally in A is lost forever as they are overwritten by the contents of the B register. There is a 'general way' of writing down such Register Addressing mode instructions. This is:

```
LD      r1,r2      load r1 from r2
```

where r1 is the destination register and r2 the source. They can be any 8 bit register except for F. Each of these instructions is represented to the CPU by a single, 8 bit, number. For example, the instruction:

```
LD      A,C     load A from C
```

is represented by the number 121. There are many such transfer commands, as you'll see if you look at the tables in the back of the book. These Register Addressing mode transfers are similar to BASIC commands of the form:

```
LET     A=B
```

They are clearly very useful, but only if we've been able to get something into the source register in the first place!

Immediate Addressing

This mode solves that problem. It allows us to carry out an operation on an 8 bit number specified as part of the instruction. In this case, the mode allows us to load a register with a given 8 bit number. This type of instruction will thus need two bytes to represent it; one for the instruction, and a second byte to represent the number in use. The first byte, as well as indicating the fact that it's an immediate mode instruction, also lets the CPU know what register is needed. The general format of these instructions is:

```
LD      r,n     load register r with value n
```

where 'r' is an 8 bit register, with the exception of F, and 'n' is the number to put in that register. A more specific example is:

```
LD      A,23
```

which will put the value 23 into the A register. This is represented by the two bytes:

```
62      23
```

The 'LD A,n' instruction is represented by the number 62, and goes in to the computer memory first, followed by the data byte, which in this case is 23. These two bytes are given special names; the instruction byte is called the **OP CODE** or **INSTRUCTION CODE**. OP CODE is short for OPeration Code. The second byte is called the **OPERAND**, which simply indicates that it is the number to be operated on by an instruction. Thus, Immediate Addressing allows us to load the CPU registers with particular values.

So far we've only looked at addressing modes that allow us to access the CPU registers. What about memory? We'll now look at addressing modes that permit us to use memory in our transfers.

Register Indirect Addressing

Things now begin to get a little more complicated, but the instructions using this addressing mode are very powerful indeed. Data is transferred between the CPU registers and memory, using the BC, DE and HL register pairs to hold the address of the place in memory which is to be involved in the data transfer. We'll look at how we can set up the 16 bit register pairs to hold the address later in the book, but for now we'll just look at the transfers possible with them. The general way of writing these instructions is:

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

Here, 'rr' indicates one of the register pairs and 'n' is an 8 bit number. There are a couple of points to note with regard to these instructions.

1. The brackets surrounding the register pair indicate that we are interested in the CONTENTS of the register pair. This is true in all Z80 operations. Whenever we see the brackets, we must remember that we're interested in the contents of the register pair or memory address that is in the brackets.
2. The HL pair is already showing its versatility over the other 16 bit register pairs. We can load a memory location whose address is in the HL register pair directly with an 8 bit number. To do the same job with the other register pairs it's 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 Immediate part of the name comes from the use of a number in the instruction ('n'), and the Register Indirect part comes from the fact that the address is held in the HL pair.

The Register Indirect Addressing instructions just seen need a single byte to represent them, with the exception of LD (HL),n, where a second byte is required to hold the value of 'n'.

Using the instructions we've seen so far, let's write a piece of machine code to transfer the contents of memory location 40000 to location 40001. Before we start, the only thing to note is that when we're using the Register Indirect Addressing instructions the high byte of the address is held in the high register of the pair (B, D or H) and the low byte of the address is held in the low register of the pair (C,E or L). The low and high bytes of the address can be worked out in the following way:

1. Calculate INT(address/256). This is the high byte of the address.
2. Calculate address - 256* (INT(address/256)). This is the low byte of the address.

Listing 4.1 shows the program to transfer the data from 40000 to 40001. In addition to the mnemonics, the decimal numbers representing the instructions are listed.

```
LD      H,156     38,156
LD      L,64      46,64
LD      A,(HL)    126
LD      H,156     38,156
LD      L,65      46,65
LD      (HL),A    119
RET               201
```

Listing 4.1

Let's look at what each instruction does. The first two load the address 40000 into the HL register pair. The low byte of the register is loaded into L and the high byte into H. Then A is loaded from the address held in the HL register pair, in this case 40000. The HL pair is then set up to hold 40001, which is the address we want to transfer the data to. The contents of A

are then written into address 40001. The final instruction, RET, brings us back to BASIC. The below BASIC program will enter the bytes into memory at address 40002. Line 10 reserves the memory, and the program is run by POKEing a number into address 40000, RANDOMIZE USR 40002, then PRINT PEEK 40001 to see if the value has been transferred. This might seem a rather trivial example, but demonstrates the basic principles of entering a small machine code program into the Spectrum 128/Plus 2.

```
 10  CLEAR 39999
 15  RESTORE 100
 20  FOR i=0 TO 10
 30  READ a
 40  POKE (40002+i),a
 50  NEXT i
 60  STOP
100  DATA 38,156,46,64,126,38,156,46,65,119,201
```

It's clear that things would be a little easier if we could have loaded the A register from a memory location without having to put the address of the memory location of interest into the HL pair. Well, the CPU does provide us with an addressing mode to do this.

Extended Addressing

These commands are of the form:

```
LD      A,(nn)
LD      (nn),A
```

where 'nn' is a 16 bit number that represents the address of a location within the computer memory. Typical instructions using this addressing mode might be:

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

These two instructions could obviously be use in listing 4.1 to transfer the data from 40000 to 40001. As you'll probably remember, the 16 bit numbers are stored in two bytes in the

memory. The instruction also needs a byte to represent it, so these instructions are three bytes long. The command:

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

is represented by the three numbers:

```
58,     64,     156
```

The address is stored in a very special way; the **LOW BYTE GOES FIRST!!** This is the case for all Z80 instructions; the CPU always expects to find the low byte of an address first, and the high byte second. The low byte goes into memory in the lowest numbered of the two memory locations, and the high byte goes into the highest numbered of the two memory locations. This may seem a little odd, but the CPU does manage to work things out. All we have to do is to work out the high and low bytes for a particular address and make sure that they go into memory in the right order. Of course, if you're using an assembler, we don't have to bother with all of this fiddling around with the order in which bytes go into memory; the assembler looks after all this for us! We saw a little while ago how to work out the low and high bytes of an address, and you can probably see that if the address is less than 256 in value then the high byte would have the value 0. Despite this, we can't just ignore it. For example, for location 255, the high byte is 0 and the low byte is 255. We have to use three bytes:

```
58,     255,    0
```

The CPU is expecting the address to be represented in two bytes, no matter what its value is. If you forget this fact with Extended Addressing, then a crash often results or, if you're lucky, an odd result! The reason for this is that the CPU will take a byte from the next instruction to 'make the numbers up', thus resulting in the rest of the program being misunderstood by the CPU.

Incidentally, the process of working backwards from a high and low byte stored in memory to a 16 bit value. Simply work out:

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

Thus, if location 40000 and 40001 are known to be holding a 16

bit number, and 40000 is holding 23 and 40001 is holding 2, the total value is given by:

```
value=2 * 256 + 23
     =512 + 23
     =535
```

Let's now return to the addressing modes. The opcodes and the operands for the instructions discussed above are:

```
LD      A,(nn)    58,     lowbyte, highbyte
LD      (nn),A    50,     lowbyte, highbyte
```

To see a concrete example of the use of Extended Addressing, the three instructions from Listing 4.1:

```
LD      H,156
LD      L,64
LD      A,(HL)
```

could be replaced by the single instruction:

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

On executing this command, the A register is loaded from address 40000. Similarily, the instruction:

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

would transfer the contents of A into address 40001.

Labels in Machine Code

When we're writing programs in assembler language, it often gets tedious to have to remember the exact addresses of locations in memory that have been used. To make life easier, we often give names to commonly used locations in memory. These names are called LABELS. Thus we might call location 40000 'FRED', and we could thus write in our assembler program:

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

As long as we remember to put the actual addresses in the program when we hand assemble the routine, we're OK. If we're assembling the program with an assembler program, then the assembler will look after this for us. If the names are

meaningful to the programmer, then the use of labels can make assembler listing more readable.

Indexed Addressing

Remember the IX and IY 'feet'? Well, here's where they come in useful. This addressing mode makes use of these registers. Indexed Addressing is used by a variety of different instructions, not just the load instructions. Indexed Addressing Load instructions are:

```
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. 'd' is an 8 bit SIGNED number; that is, it's in Twos Complement representation and represents a number between -128 and +127. It is called the DISPLACEMENT byte, and is used as follows. Assume that IX is set to hold the address of the memory location called 'tablestart' in Figure 4.1

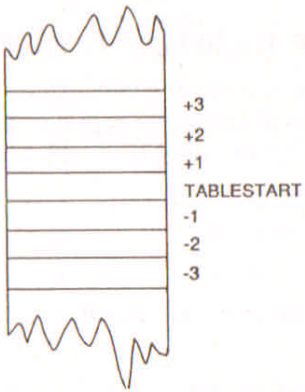


Figure 4.1

LD A,(IX+2) copies the byte from location 'tablestart+2' into register A. The displacement byte is thus a value that is added to the address held in the Index Register to get a final address which will be used by the instruction. These index registers are

thus useful when we need to get access to 'tables' of bytes in memory. The Indexed Instructions have a two byte op code, and a further byte that represents the displacement. Thus the instruction:

```
LD      A,(IX+2)
```

is represented by the numbers:

```
221,    126,    2
```

where 221 and 126 are the op code and 2 is the displacement byte. To load the A register from the address 'tablestart-1' we'd have to make the displacement byte -1. The displacement would now be 255, as the Twos Complement representation of -1 is 255. Thus LD A,(IX-1) would be represented by 221, 126, 255.

The op codes for the various indexed addressing instructions are found in the back of the book. You may remember how we could load an address with a particular number using the HL register to hold the address. Well, a similar thing is possible with the Index Registers, and the mode is 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 with a particular number. 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:

```
LD      (IX+2),34
```

which is coded as 221, 126, 2, 34; here we have a two byte op code, a displacement byte and then the data byte.

To finish the Chapter, there's a table summarising the various instructions we've seen so far. The 'bytes' column

indicates the number of bytes used to represent the instruction, Time Taken is a measure of the time needed by the CPU to execute the instruction, and Flags indicates which of the flags in the F register will be altered by the instruction. We'll look at the F register in a later Chapter, so don't panic.

Mnemonic		Bytes	Time Taken	Flags CZP/VSNH
LD	r,r	1	4	--- ---
LD	r,n	2	7	--- ---
LD	A,(add)	3	13	--- ---
LD	(add),A	3	13	--- ---
LD	r,(HL)	1	7	--- ---
LD	A,(BC)	1	7	--- ---
LD	A,(DE)	1	7	--- ---
LD	(HL),r	1	7	--- ---
LD	(BC),A	1	7	--- ---
LD	(DE),A	1	7	--- ---
LD	r,(IX+d)	3	19	--- ---
LD	r,(IY+d)	3	19	--- ---
LD	(IX+d),r	3	19	--- ---
LD	(IY+d),r	3	19	--- ---
LD	(HL),n	2	10	--- ---
LD	(IX+d),n	4	19	--- ---
LD	(IY+d),n	4	19	--- ---

Flags Notation:

\# indicates flag is altered.
\- indicates flag not changed.
0 indicates flag set to 0.
1 indicates flag set to 1.

Table 4.1 8 bit data transfers.

Chapter 5
8 Bit Counting

In Chapter 4 we examined the ways in which we could transfer data between CPU registers and the computer memory. This is all very well, but most computer programs do actually need to perform some arithmetic operations. We'll now look at the arithmetic operations that can be done by the Z80 on 8 bit numbers. We'll also look at the 'odd man out' amongst the registers, 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 arithmetic or logical operations carried out by the CPU, irrespective of whether these operations are 8 or 16 bit operations. We never treat the F register like other registers; we can't load it from memory, or operate on it with arithmetical instructions. All we can do with it is PUSH and POP it to and from the stack. Instead of treating the F register as a byte, look at it as 8 separate bits, each bit representing whether or not a certain event has occurred. These are known as **FLAG BITS**, and there are 6 of them in the F register. The other two bits aren't used; I assume that the designer of the Z80 ran out of ideas for these two bits.

There's a bit of jargon with regard to flags. We say that if it's set to a value of 1 then it is said to be **SET**. When a flag has a value of 0 it is said to be **RESET** or **CLEAR**. After an instruction has been executed by the CPU, the F register is updated by the CPU if any flags have been altered. Not all the instructions available to the CPU affect the flags; for example, the load instructions previously seen don't affect any flags.

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

We'll now look at the significance of each bit of the F register. Two things should be noted from the above, however.

1. Bits 3 and 5 are given the name 'X'. These have no relevance whatsoever to the programming or behaviour of the CPU. 'X' is used in computer circles to designate 'Don't Care'!
2. Bit 2 has a two letter name, P/V. This is because the flag is used to indicate different things by different instructions.

The C Flag

This is also called 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 and 16 bit numbers can be in the range 0 to 65535. Now, that happens in a simple sum like:

```
  11111111      In binary, 1+1=0 carry 1
+ 00000001
------------
1 00000000
```

Only the 8 lower bits (all 0) would be held in the register in use. There is no room for the ninth bit, which is set to 1. Thus the answer held in the register would be 0, rather than 256, which is what might be expected. This is simply because there aren't enough bits in the register to represent the correct result.

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 required, then the 'ninth bit' goes into the Carry Flag of the F register. A similar problem occurs in subtraction problems, like 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 Z80 arithmetic operations. It is so important, in fact, that there are two instructions that allow us to directly manipulate the C flag. These are:

SCF with op code 55

CCF with op code 63

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 its opposite state. That is, if it's set to 1 when CCF is executed the C flag will be reset to 0, and vice versa.

The N Flag

This is called the **SUBTRACT** flag. It is used mainly by the rather special BCD arithmetic instructions that we'll encounter later in this Chapter. It indicates, surprise, surprise, that the last instruction was a subtraction operation which set the flag.

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 the A register. It is used in BCD arithmetic.

The P/V Flag

This is the **PARITY/OVERFLOW** flag, the function depending upon which instructions set or cleared the flag. Let's see how it's used in both its incarnations.

PARITY

I'm sorry about all these new terms; they do come in useful for impressing people, though! Parity is a concept which turns up when we start using logical operations in the CPU. If a byte has an odd number of bits in it set to 1 it is said to have an **ODD PARITY**. If a byte has an even number of bits set to 1 then it has an **EVEN PARITY**. If A has odd parity then P/V will be set to 0, otherwise it will be set to 1. As an example of parity:

01101111 the parity is even

10000011 the parity is odd

This only indicates the parity of a byte when logical operations have been carried out on that byte.

OVERFLOW

This flag is used when we're handling Two's Complement arithmetic. It indicates that the addition of two positive numbers represented in Two's Complement notation has given rise to a **NEGATIVE** number! This isn't very likely, is it? It results from the sum being greater than 127 for 8 bit Two's Complement or greater than 32767 for 16 bit Twos Complement. It also indicates when the addition of two negative Two's Complement numbers have given rise to a positive number; again, this isn't possible. Either of these situations is signalled by P/V being set to 1.

The Z Flag

This is the **ZERO FLAG**, and is probably the most commonly used flag in the Z80 CPU. It indicates whether or not the result of a particular operation was zero. The result being tested should be in the A register, and the Z flag often indicates whether a value of 0 was left in the A register. When the result is zero, the Z flag is set to 1. When the result is NOT 0, the flag is cleared to 0. This 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 of the A

register, and so in accordance with Two's Complement notation it will be set to 1 if the result is negative and 0 if the result is positive. The flag, therefore, reflects 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 we can have something similar. We use the status of the flags, in conjunction with some instructions that we'll examine later, to form these commands. The only difference is that the instructions after the machine code IF...THEN must be the machine code version of GOTO or GOSUB. However, we're getting ahead of ourselves here. These 'IF...THEN' instructions are called **CONDITIONAL INSTRUCTIONS**; they are only executed when a certain condition is met. We'll look at them later in the book.

Counting with 8 bits

The easiest arithmetic operation that I can think of is to simply add or subtract 1 from a number held in a register. This is an easy job for the CPU, so let's see how we can do it.

Counting Up

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

```
INC    r
```

where 'r' is one of the 8 bit registers. Thus, a typical instruction might be INC A. This will increase the value held in the A register by 1. If A were to be holding 1 before the INC command, it would be holding 2 after it. When using INC, we can use other addressing modes as well. For example:

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

are both legal instructions. The address of the memory location of interest is specified by the contents of the Index Register and the displacement, as for the load operations. The two instructions above would both increment the value of the memory location addressed by a value of 1. We can also use the HL register pair to specify the address of a byte to be operated on by INC:

```
INC    (HL)
```

will add 1 to the value held in the memory location whose address is in the HL register pair. For an example, look at listing 5.1.

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

Listing 5.1

OK, so I've cheated a little here! I've used a sixteen bit load operation in the first instruction. LD HL,nn just loads a 16 bit number into the HL pair. We will look at it in detail later. However, all the other instructions have been dealt with before. Can you see what this routine will do without running it? Yes, it will result in address 40000 holding the value 1. The second and third instructions in the above routine load 0 into address 40000, and then location 40000 is incremented. To demonstrate this, try the below program. (Listing 5.2)

```
10   CLEAR 39999
15   RESTORE
20   FOR i=0 TO 4
30   READ a
40   POKE (40002+i),a
50   NEXT i
60   POKE 40000,31
70   RANDOMIZE USR 40002
80   PRINT CHR$ (PEEK(40000))
```

```
90  GOTO 70
100 DATA 33,64,156,52,201
```

Listing 5.2

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

Listing 5.3

Listing 5.3 shows the assembler listing of the bytes POKEd into memory by Listing 5.2. Run the program, and you'll see the character set of the Spectrum 128/Plus 2 printed to the screen. The printing is done in BASIC by the line 80 PRINT statement; the machine code simply increments the contents of address 40000 each time it is called.

One interesting point about INC instructions is that it's possible to get to zero by incrementing! This isn't as mysterious or as illogical as it sounds. Remember the carry flag? Well, incrementing a register that already contains the value 255 will set the 8 bits of the register to zero, thus setting the value held by the register to zero.

Counting Down

The instruction for counting down with a register or memory location are analogous to those for counting up. The instruction is called DEC, which is short for DECREMENT. The below instructions are thus all valid decrement instructions.

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

The DEC instructions all reduce the value of the register or memory location concerned by 1. As an example of the use of DEC, the below few lines of code will reduce the value of address 40000 by 1.

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

Just as you can get to zero by repeatedly incrementing a memory location or register, you can get to 255 by repeatedly decrementing a byte. As soon as the register contains the value 0, a further decrement will leave 255 in the register. This is a useful programming trick to know of in some circumstances.

Let's now go and look at the way in which these operations alter the CPU flags. By the way, INC and DEC are the first instructions that we've met that actually alter the flags.

Effect on the Flags

The 8 bit INC and DEC instructions affect most flags. As an aside, the 16 bit DEC and INC instructions, that we'll meet later in the book, don't affect the flags. Whether this was a mistake by the CPU designer, deliberate policy or an attempt to destroy the sanity of programmers we'll never know! The only important flag that is NOT affected by the INC and DEC instructions is the C flag. The notes here, remember, only apply to the 16 bit DEC and INC 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 is set if the operation altered bit 7 of the result.

HALF CARRY

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

The N flag is set if the last instruction was a subtraction. DEC can be seen as subtracting 1 from the result, so N will be set after a DEC.

It's all very well to be able to count up and down by 1s. However, for performing more complicated arithmetic operations, we're going to need some slightly more complicated operations. Let's look at these next.

8 Bit Arithmetic

In this section we'll look at the 8 bit arithmetic operations – addition and subtraction. Both of these types of operation involve the A register in a central role. Indeed, there are some instructions where the A register is taken so much for granted that it's not even mentioned in the mnemonic of the instruction! The arithmetic operations available to the Z80, you will have noticed, do not include multiplication or division. If we want these, we have to program them!

8 bit addition

The simplest add operation is:

```
ADD  A,r
```

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

```
ADD  A,B
```

The ADD A,A instruction offers us a quick way of working out twice the value of A. The 8 bit add operations can affect the value of all the CPU flags; they set the N flag to 0, and alter

the other flags in F to reflect the result of the operation just performed. The 8 bit addition operations are available in the other Z80 Addressing Modes:

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

We don't have an 'Extended Addressing' mode ADD instruction. This would be ADD A,(nn) where nn is a 16 bit address. However, this is no real loss as we can easily simulate this mode:

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

Again, we'll see the 16 bit load operation later. A very important thing to remember is that the result of an 8 bit addition must be capable of fitting in an 8 bit register. For example, adding 128 to 128 will leave the result 0 in the A register, and not the real result of the addition, which is 256. This is because 256 can't be represented in 8 bits.

However, the addition operation will set the C flag and so we'll be aware of the error that has occurred. Let's now look at a simple program to add two numbers together. I'll give you the assembler listing and the BASIC program used to call the machine code; I'll leave it to you to assemble the program, either by hand or with an assembler. (Listing 5.4)

```
10  ORG   40002
20  LD    A,(40000)
30  LD    B,A
40  LD    A,(40001)
50  ADD   A,B
60  LD    (40000),A
70  RET

10  CLEAR 40000
20  INPUT a
```

```
30  INPUT b
40  POKE 40000,a : POKE 40001,b
50  RANDOMIZE USR 40002
60  PRINT PEEK(40000)
70  GOTO 20
```

Listing 5.4

Assemble the program to address 40002. Once the machine code bytes are in place, RUN the BASIC program. Enter in a couple of small numbers in response to the prompts. The result should be printed to the screen, after the addition has been carried out by the machine code routine. The machine code listing here is as you might type it in to an Assembler program; ORG simply stands for ORiGin, which is where we want the first byte of the program to go. In this case, we want the program to start at address 40002. The two numbers are loaded into the A and B registers from addresses 40000 and 40001, then the addition is carried out. Finally, the result is stored in address 40000. Note that the result cannot handle negative numbers.

If you're the sort of person who gets a certain pleasure from seeing machines apparently making mistakes (aren't we all??), then try typing in something like 123 and 245 in response to the prompts. The correct result would be:

123 + 245 = 368

Instead, the result 112 is returned. This is the overflow problem surfacing. 112 is simply 368-256, the 256 coming from the 'lost' carry. What can we do to get around this problem? The answer is to use a new instruction called ADC, which stands for ADd with Carry. This enables us to use the carry generated in these situations.

Add With Carry

These instructions work in the same way as the normal ADD instructions, use the same addressing modes and also work on 8 bit numbers. Where they differ is that when we use ADC the

value of the carry flag, be it 1 or 0, is added to the final result. It thus enables us to make use of the carry generated by other operations. Let's take an example to make the situation clearer. Look at the addition:

100 + 200 = 300

This can only be properly represented in 16 bits. We can use the ADC instruction to add 2 8 bit numbers together that give a 16 bit result with no problems over lost carries. Listing 5.5 will, if assembled to a suitable address in memory, put the result of adding 100 and 200 into locations 40000 and 40001.

```
LD      A,200       ; A=200
LD      B,100       ; B=100
ADD     A,B         ; add the numbers
LD      (40000),A   ; A now holds low
                    ; byte of result.
                    ; store it in 40000
LD      A,0         ; as we've got 8 bit
LD      B,0         ; numbers the high
                    ; bytes are 0, so
ADC     A,B         ; use ADC-this will
                    ; carry out the sum
                    ; of A + B + carry
LD      (40001),A   ; store the high
RET
```

Listing 5.5

The contents of the two addresses can be checked with:

```
PRINT PEEK(40000) + 256* PEEK(40001)
```

The ADC operation takes the carry flag value, in this case 1, and adds it to the two zeroes in the A and B registers. As was mentioned in the listing, we treat the two 8 bit numbers that we started with, 100 and 200, as 16 bit numbers with high bytes of zero. These high bytes are then used by the ADC instruction to add the carry, and so make the high byte equal to 1. Listing 5.6 shows a more general program which adds the contents of address 40002 to the contents of 40003 and stores the result in address 40000 and 40001.

```
LD      A,(40002)
LD      B,A
LD      A,(40003)
ADD     A,B
LD      (40000),A
LD      A,0
LD      B,0
ADC     A,B
LD      (40001),A
RET
```

Listing 5.6

Try this program out. The values to be added should be POKEd into addresses 40002 and 40003. They should clearly be 8 bit numbers. Use RANDOMIZE USR address to call the routine, then print out the contents of addresses 40000 and 40001.

8 Bit Subtraction

The instructions for 8 bit subtraction operations are similar to the 8 bit addition operations. Again, there are two different types of subtraction operation, the subtract without and the subtract with carry. These are:

SUB subtract without carry

SBC subtract with carry

These operations have the same addressing modes as the addition operations. The result of the subtraction is left in the A register. The SBC operation subtracts the value of the C flag as well as the value of the operand.

Before leaving the subject, another useful trick is that the SUB A,A instruction will have the result of setting the A register to zero. The SUB A,A instruction is also a method of clearing the C flag. It's very important in the ADC and SBC instructions to keep an eye on the status of the C flag.

The final point is one of mnemonics. Certain of the instructions that we've just examined are occasionally written without even 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 instructions, you'll know exactly what they mean. Some of the assembler routines will accept BOTH these sets of mnemonics, others just one or other.

BCD Arithmetic

So far, all the arithmetic that we've seen has been straightforward binary arithmetic. Now's the time to introduce a different type of arithmetic, called **Binary Coded Decimal** arithmetic, or **BCD** arithmetic. It's also a new way of representing numbers. However, it's not very commonly used, and I'm including it here only for the sake of completion. It's often used when we need to communicate between the computer and some peripherals, particularly scientific instruments. However, let's look at how BCD works.

Rather than treat the 8 bit byte as a single number, BCD treats it as two 4 bit nibbles, and each nibble represents a digit between 0 and 9. The high nibble, though, has ten times the significance of the low nibble, just like the tens and units columns in normal arithmetic. It's clear from this that 6 of the 16 possible different combinations of 4 bits aren't needed – 0 to 9 only uses 10 of the combinations. In BCD, therefore, a byte represents a two digit number between 0 and 99. To make this clearer, look at the below examples.

0010	0010	22 in BCD, 34 in binary
1001	1001	99 in BCD, 153 in binary
1101	1101	221 in binary
		ILLEGAL IN BCD.

Arithmetic with BCD

The Z80 allows us to perform arithmetic operations on BCD numbers, but can cause a pot full of problems for the unwary. Consider the below sum:

```
  0000 0011     3
+ 0000 1000    +8
-------------  ---
  0000 1011    11
```

This is correct in binary, but not for BCD. The number 1011 is illegal as a BCD nibble. The binary results of such calculations must be modified in some fashion to get a legal BCD representation of the binary number. We thus do the arithmetic in binary, then convert the result into BCD. We can get a proper BCD number by adding 6 to the number **PROVIDED THAT** the low nibble has a value greater than 9. Fortunately, we don't have to write programs to do this ourselves; the CPU designers put in an instruction that converts a binary byte into a BCD byte. The instruction is called DAA, which stands for **D**ecimal **A**djust **A**ccumulator. Any carry from the low nibble to the high nibble caused by this operation is indicated by the H flag.

Comparing Numbers

What is a section on comparing numbers doing in a chapter on 8 bit counting? Well, the process of comparing two numbers for the Z80 is essentially one of subtracting the two numbers from one another. If you think about it, there are three possible results of such a subtraction. A result of zero will indicate that the two numbers are the same. A negative or positive result indicates that one number is larger than the other. Rather than use a subtraction operation, which would alter the value of the A register, the Z80 is equipped with a special instruction called **CP**, which stands for **C**om**P**are. It again works in a variety of addressing modes but always compares the value in the A register to that held in another register or memory location. The addressing modes available are:

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

If we examine the CP instruction in more detail, then we find that a 'subtraction' is carried out which subtracts the contents of the specified register or memory location from the contents of the A register. However, and very importantly, the result of this is NOT put in the A register, thus preserving its contents. CP does not affect the values of any registers, but it does alter the status of the flags in the F register. Although the A register is not mentioned by name in the above instructions, it is always assumed by the CPU. CP B,C instructions are not, therefore, available. The N flag is always set by a CP instruction, because the operation is essentially a subtraction. The other flags are set according to the result of the subtraction, and we can then use these flags to control what happens next in our machine code program. We'll look at the conditional instructions, whose execution depends on the status of one of the flags, in a later chapter. For the CP instruction, the P/V flag is used to indicate Overflow.

The two flags most commonly used when we're testing the result of a CP operation are Z and C. Let's take an example:

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

Here, CP B will compare the contents of the B register to the contents of A, which in this case is 10. 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've thus got three unique situations, and so by testing the Z and C flags after a CP operation we can see what the relative magnitudes of the numbers were. A couple more examples are:

```
LD    A,22
LD    B,21
CP    B

Z=0,  C=0
```

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

Z=0, C=1

Again, it's only by testing the flags that we can see what the result of a comparison was.

Mnemonic		Bytes	Time Taken	Effect on Flags CZP/VSNH
ADD	A,r	1	4	### #0#
ADD	A,n	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,r	1	4	### #0#
ADC	A,n	2	7	### #0#
ADC	A,(HL)	1	7	### #0#
ADC	A,(IX+d)	3	19	### #0#
ADC	A,(IY+d)	3	19	### #0#
SUB	A,r	1	4	### #1#
SUB	A,n	2	7	### #1#
SUB	A,(HL)	1	7	### #1#
SUB	A,(IX+d)	3	19	### #1#
SUB	A,(IY+d)	3	19	### #1#
SBC	A,r	1	4	### #1#
SBC	A,n	2	7	### #1#
SBC	A,(HL)	1	7	### #1#
SBC	A,(IX+d)	3	19	### #1#
SBC	A,(IY+d)	3	19	### #1#
CP	r	1	4	### #1#
CP	n	2	7	### #1#
CP	(HL)	1	7	### #1#
CP	(IX+d)	3	19	### #1#
CP	(IY+d)	3	19	### #1#

Flags Notation:

flag is altered.

0 flag set to 0.

1 flag set to 1.

- flag not altered.

Table 5.1 Arithmetic Operations

Logical Operations

Strictly speaking, of course, all the operations carried out by the CPU are logical! However the operations I'm referring to here are instructions which work on the bits within a byte rather than the byte as a whole. The instructions in this category can be very valuable to the machine code programmer. The bit oriented operations are often called **BOOLEAN** operations, after the Dublin mathematics professor who first described them, long before computers were thought of! Just as addition and subtraction are said to be arithmetic operators, these are Logical, or Boolean Operators. There are 4 Boolean Operators that are supported by the Z80 CPU. These are:

AND
NOT
OR
XOR

Let's now look at each of these in turn and see what they do.

Truth Tables

A quick diversion here; just as we have tables to describe the results of multiplication operations on normal numbers, we have tables that describe the effects of applying logical operations to bits. These tables are called **TRUTH TABLES.** Each of these operations has a different Truth Table, as we shall now see.

The NOT Operation

This works on just 1 bit. It's effect is:

A	NOT	A
0		1
1		0

You'll probably remember that this is the effect of the Complement operation, like the one we saw when we looked at Two's Complement numbers. There isn't a NOT command in the Z80 instruction set; it's called CPL, which stands for ComPLement, instead. Also, it works on the whole of the A register, converting each 0 to a 1 and each 1 to a 0. There are no other addressing modes for the CPL instruction; it can only operate on the A register. One application of the CPL instruction is to generate Two's Complement numbers:

```
CPL
INC  A
```

The AND Operation

The AND operation is available in a variety of addressing modes, and works on 2 bits. The addressing modes are:

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

Note how the A register isn't mentioned at all here. The AND operations all involve the A register as the source of one set of bits. The operation works on corresponding bits from the A register and another register or memory location, and combines them in accordance with the truth table below. The result is left in the A register.

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

The AND operation only gives a 1 as result if both bits involved are also set to 1. To give a more concrete example, the command A AND B will do an AND operation on the bits in the A and B registers. The result of the operation on a particular set of data is as follows:

```
A         00001010
B         00101101
A AND B   00001000
```

or, we might have:

```
A         10011111
B         01101110
A AND B   00001110
```

We can thus use the AND instruction to MASK OFF certain bits of a byte that we're not interested in by setting the corresponding bits of the other byte to 0. Then, it doesn't matter what the first bits were set to; ANDing them with zero will always return zero. For example, to ensure that a byte only has a value between 0 and 15, we could simply get rid of the top 4 bits. We could do this as follows. Address 40000 holds the value to be 'fudged' to be between 0 and 15.

```
LD      A,(40000)
AND     15              ; 15 is binary 00001111
LD      (40000),A
RET
```

No matter what value is in the top 4 bits of address 40000, it will be totally disregarded. This gives us another useful programming trick; setting a particular bit of a byte to 0. All we do is AND the byte with a value that has the corresponding bit set to 0. So to set bit 4 of A to zero, we'd just AND it with 223, which is 11101111 in binary.

The OR Instruction

The truth Table for OR is:

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

This operation gives a result of 1 if either of the bits involved is set to 1. The addressing modes are the same as for the AND instruction, and we can only OR with something in the A register. Again, the result is left in the A register. We can use OR to set bits within a byte to 1 in a similar way to that in which we used AND to set a bit to 0. So to set bit 1 of A to 1, we could use:

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

Of course, the OR 2 instruction would also do the job.

Note that although the AND and OR instructions are present in Plus 2 BASIC, they do not operate on a bytes in this fashion; they simply allow us to combine true and false statements. The logical operators are often called BITWISE operators, 'cos they work on bits.

The XOR Operation

XOR is not a character from a science fiction novel, though if I ever write one he, she or it may well be in there!! XOR is short for EXclusive OR, and its truth table is:

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

As you can see, a result of 1 is only obtained when either one of the bits is 1, BUT NOT both of them. This is where the exclusive OR comes in. Alternatively, look at XOR as a function that

returns a value of 1 when the bits involved have different values. The addressing modes available to XOR are the same as those available for AND or OR. One use of the XOR instruction is to set the A register to zero; XOR A will do this and needs only a single byte instruction to do it.

Effect on Flags

As can be seen from the following table, all the flags are affected in some way by these commands. However, only three flags have a status that is directly affected by the operation. These are:

Z. This is set if the result is 0.

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

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

The Boolean operations can also be used to manipulate the C flag status. OR A will leave A intact but clear the C flag. XOR A will clear the C flag and set A to 0.

Mnemonic		Bytes	Time Taken	Effect on Flags CZP/VSNH	
AND	r	1	4	0##	#01
AND	n	2	7	0##	#01
AND	(HL)	1	7	0##	#01
AND	(IX+d)	3	19	0##	#01
AND	(IY+d)	3	19	0##	#01
OR	r	1	4	0##	#00
OR	n	2	7	0##	#00
OR	(HL)	1	7	0##	#00
OR	(IX+d)	3	19	0##	#00
OR	(IY+d)	3	19	0##	#00
XOR	r	1	4	0##	#00
XOR	n	2	7	0##	#00
XOR	(HL)	1	7	0##	#00
XOR	(IX+d)	3	19	0##	#00
XOR	(IY+d)	3	19	0##	#00

Flags Notation.

# indicates flag changed.

0 indicates flag set to 0.

1 indicates flag set to 1.

- indicates flag unchanged.

Table 5.2 Logical Operations.

Manipulating bits within a byte

There are a couple of commands in the Z80 instruction set that allow us to manipulate the status of a single bit within a memory location or register. These are called SET and RESET. However, much of what can be done with these instructions can also be done with the logical operators just discussed.

The **SET** instruction forces the value of a given bit to 1. It operates in the following addressing modes:

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

where n is the number of the bit within the byte of interest, between 0 and 7, and r is an 8 bit register. Thus the commands:

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

will result in the A register holding the value 1. No changes are made to any of the flags. The instruction that performs the reverse of this operation is **RESET**. This instruction forces the value of the specified bit to 0. It operates in the same addressing modes as SET. Thus the instructions:

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

will result in the A register holding the value 254.

It's also possible to test the status of a particular bit within a byte; a sort of single bit CP instruction. We can tell whether the bit is set to 0 or 1. It also functions in the same addressing modes

as SET or RESET. The result of the comparison, carried out by the **BIT** instruction, reflected in the status of the Z flag. Look at the following example:

```
LD    IX,2000
BIT   0,(IX+0)
```

I'm using a 16 bit load instruction to put a value into the IX register. These two instructions will look at the status of bit 0 of location 20000. If this bit is set to 0 then the Z flag will be set. If the bit is set to 1 then the Z flag will be clear.

Rotates and Shifts

These instructions aren't often used in machine code programming, but when we do use them they can be very valuable indeed. They are used to move bits around within a byte. A **ROTATE** will shift bits through a byte, and put any bits that fall out of one end of the byte into the other end of the byte. This cyclic operation is shown in Figure 5.1.

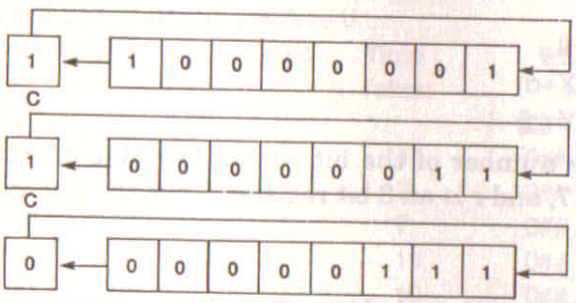


Figure 5.1. Rotates

Shifts, on the other hand, move bits through a byte but any bits shuffled out of the byte are lost forever. This is shown in Figure 5.2.

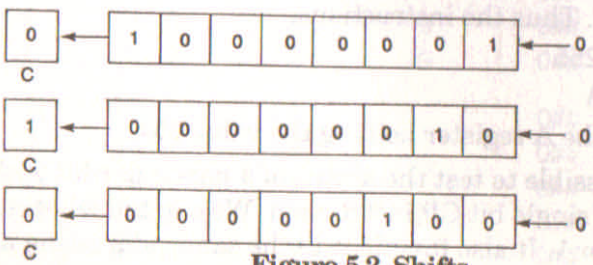


Figure 5.2. Shifts

Rotate or Shift operations are named according to the direction in which the bits are moved. If the bits are moved from left to right then we have a shift right. If the movement is from right to left we have a left shift.

Left Operations

There are two different Rotate Left operations and one left shift operation.

Rotate Left

This operation works in the following addressing modes:

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

An example is RL A which operates on the A register. This stands for Rotate Left Accumulator, and the action of the command is shown in Figure 5.3.

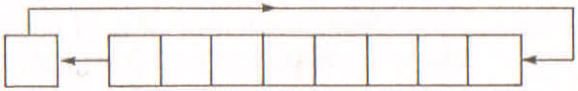


Figure 5.3. RL A

The current value of the C flag is moved into bit 0 of the A register. Bit 7 of the register is moved into the C flag. The C flag appears to act as a 'ninth bit' for the operation.

Rotate Left Circular

The RLC instruction works in the same addressing modes that were featured above. The mnemonic is RLC. For example, legal instructions 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 byte concerned.

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 that we've mentioned already. Its operation is shown in Figure 5.4

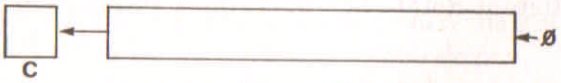


Figure 5.4 SLA

This can effectively be seen as a 'multiply by two' instruction. However, if the value in the register affected by the SLA instruction is greater than 127, then 'funny' results might be found! Listing 5.7 shows the SLA instruction in use. Put the program in a suitable location, and use it by POKEing a small value into address 40000 and then using PRINT USR address, where 'address' is the address of your routine. Remember that USR executes a program, it returns to BASIC with the value in BC as its 'result'.

```
LD      HL,40000
SLA     (HL)
LD      B,0
LD      A,(HL)
LD      C,A
RET
```

Listing 5.7

Again, my apologies for the 16 bit load.

Right Operations

The simplest operations are the Rotate Right instructions. These operate in the following addressing modes:

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

The operation is shown in Figure 5.5.

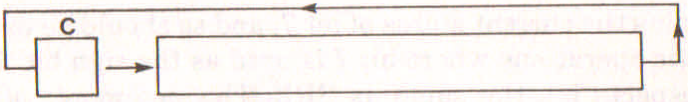


Figure 5.5 RR instructions.

Rotate Right Circular

The same addressing modes are supported as for the RR instructions, and the operation is shown in Figure 5.6. Bit 0 is copied into the C flag, but is also shifted into bit 7. The mnemonic is **RRC** s, where s is one of the available addressing modes.

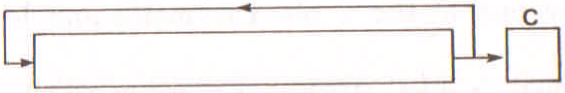


Figure 5.6 Rotate Right Circular.

Shift Right Logical

The addressing modes supported for this operation are the same as for RRC. The operation consists of shifting the contents of a byte to the right, bit 0 going into the C flag and bit 7 being replaced by a 0. You can probably see that this is a 'divide by two'. (Figure 5.7). The mnemonic is **SRL**.

Figure 5.7 SRL

One problem with using this operation as a 'divide by two' is that bit 7 is replaced by zero. If you remember what we said about Two's Complement representation of numbers, bit 7 is used to indicate the sign of a number. If a number were to be in this representation and representing a negative number, then we'd be in trouble if we used this instruction as the sign of the number would be lost. What we need for this kind of operation is an instruction that preserves the status of bit 7. Well, SRA does this.

Shift Right Arithmetic

This retains the current status of bit 7, and so should be used in arithmetic operations where bit 7 is used as the sign bit. In all other respects it's the same as SRL. The mnemonic of this instruction is SRA s, where s is one of the available addressing modes. Figure 5.8 shows the operation.

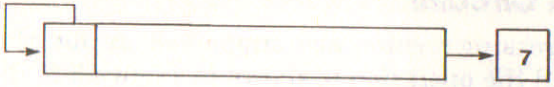


Figure 5.8 SRA

Those of you still with me will be gratified to know two things.

1. That's the end of the 8 bit arithmetic and logical operations.
2. There aren't as many 16 bit arithmetic and logical operations!

Let's now look at the 16 bit operations in more detail.

Chapter 6
16 bit data transfers

We've already seen how some of the 8 bit registers can be paired up to make 16 bit register pairs. Indeed, we've used some of these register pairs in our programs. So, now's the time to look at the instructions that are available to us to handle the 16 bit register pairs.

Let's start with a look at how we can load the register pairs with a 16 bit number. The general mnemonic for these instructions is:

```
LD      rr,nn
```

where 'rr' is a register pair, BC, DE, HL, IX or IY and 'nn' is a 16 bit number. A typical example is:

```
LD      HL,40000
```

The instructions need at least three bytes to represent them; a one or two byte op code and a two byte operand, in this case representing the number 40000. The two byte op codes are needed for instructions using IX and IY; BC, DE and HL operations use a single byte op code. As we've already seen, the number to be loaded into the register pair is stored in memory with its low byte first. Thus the instruction:

```
LD      HL,515
```

would be stored in memory as:

```
33,     3,     2
```

33 is the op code for LD HL,nn, 3 the low byte of 515 and 2 the high byte. Of course, if we're using an assembler to enter the program then we don't need to worry about this; the assembler will do it for us. The instructions for the other registers are similar, e.g.

```
LD    BC,3453
LD    IX,20
LD    DE,65535
LD    IY,0
```

Again, all numbers that are to be loaded into the register pairs, whether they be 8 or 16 bit numbers, must be represented as 16 bit numbers. So, 0 would be represented as two zeroes, 1 as a 0 and 1 and so on.

It's also possible to transfer information between registers and addresses. The instructions are of the form:

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

Because we are moving 16 bit numbers around, it's clear that there must be another byte involved, as well as that specified by address 'nn'. This is address 'nn+1', and so with the instruction:

```
LD    HL,(40000)
```

will load the register with the contents of addresses 40000 and 40001. The L register will receive the contents of address 40000 and the H register will automatically receive the contents of address 40001. Similarily, when we store the contents of a register in a two byte memory location, the lower numbered location gets the low byte of the register pair and the higher numbered location gets the high byte of the pair.These instructions have either a 1 or two byte op code.

Mnemonic		Bytes	Time Taken	Effect on Flags CZP/VSNH	
LD	rr,nn	3	10	---	---
LD	IX,nn	4	14	---	---
LD	IY,nn	4	14	---	---
LD	(nn),rr	3/4	20/16	---	---
LD	(nn),IX	4	20	---	---
LD	(nn),IY	4	20	---	---
LD	(nn),rr	3/4	20/16	---	---
LD	(nn),IY	4	20	---	---
LD	(nn),IX	4	20	---	---

Flags Notation:
# indicates flag changed
- indicates flag not changed
1 indicates flag set to 1
0 indicates flag set to 0

Table 6.1 16 bit loads.

Manipulating the Stack

Early in this book we mentioned the Stack, a place where the CPU can store pieces of information in such a way that it doesn't have to keep an eye on where it was put and so that it's conveniently available. We can also use the stack to store numbers, but we can only store 16 bit numbers on it. Information is stored on the stack by **PUSH** instructions, and is retrieved with **POP** instructions. The full set of instructions are:

PUSH	AF	POP	AF
PUSH	BC	POP	BC
PUSH	DE	POP	DE
PUSH	HL	POP	HL
PUSH	IX	POP	IX
PUSH	IY	POP	IY

Note how the F register is paired up with the A register to produce a 16 bit register pair that can be saved on the stack. PUSH copies the relevant register pair's contents on to the stack, the registers still holding their contents. POP has the effect of removing a number from the stack entirely. Also, anything in the register pair into which the information is 'POPped' is lost. POP functions on the last item in the stack only. Thus the instruction POP HL will take the last item from the stack and put it into the HL register pair.

One use of the stack is to allow us to swap information between the 16 bit register pairs. Instructions such as LD BC,HL aren't allowed in the Z80, so we normally solve such a problem with the instructions:

```
LD      B,H
LD      C,L
RET
```

or something similar. However, we could also use the instructions:

```
PUSH    HL
POP     BC
```

The register pair into which infomation is POPped doesn't have to be the same as that from which it was PUSHed. We can also look at the contents of the F register directly, without checking flags. This is done by:

```
PUSH    AF
POP     BC
```

The C register will now hold a copy of the contents of the F register, and we can now inspect it directly. The final application which we'll discuss here for the stack is to temporarily store registers whilst we're doing something else. For example,

```
PUSH    AF
PUSH    HL
  .....
POP     HL
POP     AF
```

might be used to store the contents of AF and HL while we executed some more instructions between these two. No matter what the intervening instructions did to the HL and AF contents, we could get the original contents back with no difficulty, just by POP instructions.

Warning!

Before getting too carried away by the uses of the stack, remember that it's also used by the BASIC Interpreter of the Spectrum. Also, use of the extra 'pages' of memory in the 128/Plus 2 can temporarily take the stack out of the memory map! We'll look at this later in the book. However, if you alter the stack, each PUSH should be balanced by a POP. The stack, when we enter one of our own machine code programs, is used by the CPU to remember where it was when the USR instruction was executed to run our own machine code. Therefore, this address should be the 'top' item on the stack when we finish our machine code programs with a RET, so that we can go back to BASIC properly. If it isn't, then you will almost certainly have a crash. Once you get more experience, you'll know when and how to 'break' this rule. But for now, don't! For example, a program such as:

```
PUSH  BC
RET
```

would cause problems by putting the BC register contents onto the stack. This would cause the CPU to think that the BC contents was the address to go back to when it executed the RET, and a crash would result unless the BC contents had been the same as the 'go back' address. A POP on its own would cause similar problems by removing this address from the stack altogether. However, both:

```
PUSH  BC
POP   BC
RET
 and
POP   BC
PUSH  BC
RET
```

are OK, as the 'go back' address would be at the top of the stack when the RET instruction was executed. This 'balancing' of PUSHes and POPs is rather important.

Moving the Stack

Most of the time we use the stack without knowing where it is in memory. However, the CPU, when running the Spectrum 128 BASIC Interpreter program, must set up the stack's position in memory as one of the first things it does. It does this by loading the 16 bit address of the stack into a special register called the Stack Pointer, SP. The commands available to us to alter SP are:

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

Obviously, moving the stack around in memory isn't very advisable under most circumstances. Leave this sort of activity until you've got a little experience under your belt. Before leaving Stack operations, Listing 6.1 shows a routine that returns to you the address on the stack when the routine was called from BASIC. Note how we use the BC register pair to return the result to BASIC, and how we ensure that POPs and PUSHes balance up. The routine can go to any address in memory that you want. Use the tables in the back of the book to get the Op Codes for PUSH BC and POP BC. Table 6.2 features the stack operations.

```
POP   BC    ; get address
PUSH  BC    ; copy back to stack
RET         ; all done!
```

Listing 6.1

Mnemonic		Bytes	Time Taken	Effect on Flags CZP/VSNH
PUSH	rr	1	11	--- ---
PUSH	IX	2	15	--- ---
PUSH	IY	2	15	--- ---
POP	rr	1	10	--- ---
POP	IX	2	14	--- ---
LD	SP,nn	3	10	--- ---
LD	SP,(nn)	3	20	--- ---
LD	SP,IX	2	10	--- ---
LD	SP,IY	2	10	--- ---
LD	SP,HL	1	6	--- ---

Flags Notation:

\# indicates flag altered.
\- indicates flag not changed.
1 indicates flag set to 1.
0 indicates flag set to 0.

Table 6.2 Stack Operations.

The final couple of instructions are again of limited use. They are concerned with the Alternate Register set, and the Alternate Registers are used in part by the Spectrum 128/Plus 2 BASIC Interpreter. So I'll mention them in passing but suggest you take care with them!

EX AF,AF' swaps the contents of AF and A'F'. EXX does the same thing for BC and BC', DE and DE' and HL and HL'. In addition, the command EX DE,HL swaps the contents of the main register pairs HL and DE. It's a single operation that does the same as:

```
LD      (temp), HL
PUSH    DE
POP     HL
LD      DE, (temp)
```

As a final point, you may have noticed that IX and IY instructions take longer to execute than the corresponding HL ones. This is purely due to the fact that they have two byte op codes rather than just single byte op codes. The CPU has to get both op codes from memory before it knows what to do next.

Chapter 7
16 bit arithmetic and counting

The 16 bit arithmetic instructions allow us to perform various operations on the 16 bit register pairs. Although it's possible to do 16 bit addition or subtraction using the 8 bit operations that we've already seen, it's a lot easier to use the instructions that the CPU designer gave us. However, the 16 bit operations aren't as versatile as the 8 bit operations.

The simplest operations that we can perform are the 16 bit INC and DEC instructions.

INC and DEC

The INC and DEC operations that are available to us are as follows:

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

Thus the two instructions:

```
LD    HL,0
INC   HL
```

will result in the HL register pair holding the value 1. Just as with the 8 bit INC and DEC instructions, we can get to zero by repeatedly INCrementing a register pair, but we instead of going from 255 to 0 we go from 65535 to 0. Any 'carry' between the MSB of the low register and the LSB of the upper register, or a borrow for DEC instructions, is taken care of automatically by the CPU. Also, repeatedly DECrementing a register pair will eventually reach 65535. (0 - 1 = 65535).

One real problem with the 16 bit INC and DEC instructions, though, is that they **DO NOT** affect the flags! We thus have to use more instructions to check the value of the flags after such an operation. We commonly need to find out whether such a register pair contains the value zero, particularly if we've been using the register pair to control the number of times a sequence of machine code instructions was executed. (A machine code 'FOR...NEXT'). The below program section shows how we can do this:

```
DEC   HL
LD    A,L          ; get low byte
OR    H            ; OR with high byte
JP    Z,address    ; only if both are
                   ; 0 will we get a 0
                   ; result
```

The JP Z,address instruction is a Jump instruction, which passes control of the computer to another part of the program; it's a little like the BASIC GOTO instruction. We'll look at it in the next chapter. However, here it checks the Z flag, which will have been affected by the result of the OR operation, and only jumps to the address 'address' if the result is zero. This will only be so if both H and L contain 0.

Note that there are none of the other addressing modes available. We cannot, for example, directly decrement two byte memory location in the same way that we could decrement a single byte memory location using DEC (HL) for 8 bit operations. We have to load the value into a register pair, decrement the pair and then write the result back to memory instead.

Addition and Subtraction

Just as the A register is the favourite register for 8 bit addition and subtraction, the HL pair is the favourite register for 16 bit operations. The ADD instructions available are:

```
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
```

A couple of points to note here; there is no command to add HL to either of the Index Registers. In addition, there's no ADD IX,IY instruction. Also, there is no instruction of the type ADD HL,nn. If we wish to do this, we have to use instructions as follows:

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

In all these operations, the result is left in the first register pair mentioned in the instructions. So, in the instruction ADD HL,DE, the result of the addition is left in the HL register pair.

Effect on Flags

There aren't many flags altered; C is changed if there's a carry from the 7th bit of the High register of the pair to the '17th bit'. Any carry from bit 7 of the low register to bit 0 of the upper register is automatically taken care of. The N flag, as we might expect, is set to 0 by the addition.

There are no 16 bit equivalents of the SUB instructions that were available for 8 bits. If we wish to perform 16 bit subtraction operations then we need to use SBC instructions.

Add and Subtract with Carry

There is a selection of ADC instructions available to us, which, as with the 8 bit instructions, offer us the opportunity to carry out multiple byte addition. Using ADC instructions we could add together 4 or 5 byte numbers if we wanted to, just using the ADC instructions to keep an eye on the carry generated by each addition. The ADC operations available are:

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

There are no instructions to handle the IX or IY registers here.

The SBC instructions are analogous to the ADC instructions just discussed, and they work on the same register pairs. They affect the flags in the same way, but with the exception that the N flag is set to 1, indicating a subtraction operation.

Because these operations include the C flag in their result, always remember to clear the C flag to 0 unless you specifically want the C flag included in the result. The only time that this might occur with these 16 bit operations is where we've already added something together and we're wanting to include the carry generated from that operation, or the borrow taken for a subtraction operation, in the sum now in hand. The easiest way to clear C is to use a Boolean operation. Listing 7.1 shows a simple program to subtract the 16 bit number in locations 41002 and 41003 from the 16 bit number in locations 41000 and 41001. The result is left in the BC register for the return to BASIC, and you could thus use the routine by POKEing in suitable values to the addresses mentioned, then executing a PRINT USR address instruction, where 'address' is the location in memory of the routine.

```
LD      HL,(41000)
LD      DE,(41002)
AND     A               ; clear C
SBC     HL,DE
PUSH    HL              ; result into BC
POP     BC
RET
```

Listing 7.1

Mnemonic		Bytes	Time Taken
ADD	HL,rr	1	11
ADD	HL,SP	2	11
ADC	HL,rr	2	15
ADD	IX,rr	2	15
ADD	IX,SP	2	15
ADD	IX,IX	2	15
ADD	IY,rr	2	15
ADD	IY,SP	2	15
SBC	HL,rr	2	15
SBC	HL,SP	2	15

Table 7.1 16 bit arithmetic.

Chapter 8
Jumps, Loops and Block Operations

This Chapter covers two apparently different groups of Z80 instructions. These are:

1. Instructions that cause control of the CPU to pass from one part of the program to another. These are similar to the GOTO and GOSUB instructions in BASIC.
2. The Block instructions, which are operations which work on blocks of memory rather than just single bytes within memory.

The connection between these two sets of instructions is that Block Operations often involve a 'hidden' jump instruction of some sort, so it makes sense to group them together.

Jumps

The simple programs that we've entered into the 128/Plus 2 so far have all operated without the presence of any machine code equivalents of GOTO or GOSUB. Each instruction was executed in strict order, and no groups of instructions were executed two or more times. Obviously, if we were to carry on this programming philosophy then the resultant programs wouldn't be very powerful!

The ability of the CPU to execute these jump instructions thus gives us great programming power, but will also cause us to take more care in our programming. So, let's begin by looking at the **JP** instructions, or **JumP** instructions. These are the machine code equivalent of GOTO, but instead of jumping to line numbers we jump to particular addresses in the computer memory.

The JP instruction has two addressing modes; immediate and Register Indirect. In the immediate mode, the address to which the jump is to be made is stored as part of the instruction. For example,

```
JP      40000
```

will cause the CPU to jump to address 40000 and start executing whatever instructions it finds there. In the Register Indirect addressing mode, the address to which the jump is to be made is stored in the HL, IX or IY register pairs. We'll look at these instructions later in the Chapter.

The sort of jump shown above is called an **UNCONDITIONAL** jump, because the CPU will jump no matter what. As soon as the JP instruction is executed, the address to which the jump is to be made is stored in the Program Counter of the CPU and the CPU continues executing instructions from that address. Alternatively, the jump can be **CONDITIONAL**, where the JP instruction is not always executed. The jump takes place only if some particular condition, indicated by the status of one of the flags in the F register, is satisfied. This is clearly the machine code equivalent of:

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

in BASIC. For example, the instruction:

```
JP      Z,40000
```

will cause a Jump to address 40000 ONLY IF the Z flag is set. (i.e. the result of the last operation to affect the flag was zero.) Other flags can also be used in a similar fashion, such as:

```
JP      C,40000
```

which causes a jump only if the C flag is set. Other instructions of this sort are as follows.

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 is parity even
JP	PO,address	jump if parity odd

All these instructions are three bytes long. This is broken down into a two byte address and a single byte op code. The address is stored, as we might expect, in the usual Z80 format of LOW BYTE FIRST. This is very important to remember; if you put the address into memory in the wrong order the CPU will jump to the wrong address! Thus the JP 40000 instruction that we saw above is assembled to give the following three bytes:

```
195   64,156
```

195 is the op code for JP nn, and 64 and 156 are the low and high bytes of 40000 respectively.

Let's now write a simple routine that demonstrates the JP command. A word of advice before we start, though. Like the good soldier, the CPU 'jumps' when we tell it to jump with a JP instruction, even if the orders we've given are pretty silly. For example, a mistake in specifying the address could cause the CPU to jump to a byte that, instead of holding the first byte of an instruction to execute, might hold a byte of data! This could cause the computer to 'crash' or do something totally unexpected. One particular 'unexpected' event is that the CPU can start executing a never ending loop. The normal 'Break' key won't work from machine code, so you're stuck forever in the loop. The only way out is to turn the computer off! So, take care in giving addresses. Of course, if you've got an assembler then the problem isn't as big, because the assembler will look after putting the address bytes in memory.

Listing 8.1 shows a routine that fills each byte in the screen memory with the value 255. There are other ways of doing this, but this one will do for now.

```
        ORG     40000
        LD      HL,16384        33,00,64
        LD      BC,6144         1,00,24
LP1:    LD      (HL),255        54,255
        INC     HL              35
        DEC     BC              11
        LD      A,B             120
        OR      C               177
        JP      NZ,LP1          194,70,156
        RET                     201
```

Listing 8.1 Screen filling.

I've included the bytes in this listing to demonstrate how the JP instructions are coded within a program. If you use these bytes, then they MUST be loaded to address 40000 because the JP instruction passes control to label LP1, which is at address 40006. This address is given in the JP instruction. The program, when called, loads the 6144 bytes of memory that make up the screen with the value 255. We decrement the BC register pair each time, and do a check to see if it's yet to zero yet. If it hasn't, then JP NZ,LP1 takes control back to the beginning of the loop and loads the next byte of memory with 255.

You could try the below program in BASIC as a comparison of the speeds of machine code and BASIC. I'm sure that you'll be impressed with machine code!

```
10  FOR I=16384 TO 22528
20  POKE I,255
30  NEXT I
```

As a further example of machine code jumps, Listing 8.2 shows a machine code program that can produce a delay time delay- useful for slowing things down a little.

```
        ORG     40000
        LD      HL,65535
LP1     LD      DE,100
LP2     DEC     DE
        LD      A,D
```

```
OR      E
JP      NZ,LP2
DEC     HL
LD      A,H
OR      L
JP      NZ,LP2
RET
```

Listing 8.2 Time delay.

The 'DE' loop is executed, in this example, 65535 times. HL thus specifies the 'coarse setting' of the time and DE the fine tuning of the timing. For longer time delays, the inner loop using DE can be replaced with a HALT command, which causes the Spectrum 128/Plus 2 to stop until it's interrupted. In the Spectrum, these interrupts (see Chapter 10) occur once every 1/50th second, and so the HL register would then specify the delay in fiftieths of a second. Again, we have to use a short sequence of instructions to see if a 16 bit register is holding the value 0.

Once such a program is assembled to a particular address, it will only work properly at that address. If we loaded the bytes to another address, the program wouldn't work properly because the JP instructions would still cause a jump to the addresses specified when we originally assembled the program. We'll shortly see a way around this. In addition, we have to specify a full two byte address even if the address to which we want to jump is only a few bytes away from the jump instruction. Well, there are instructions that get around this problem and allow us to specify the address to which we want to jump with just a single byte. They also allow us to get around the first problem of having to re-assemble the program to get it to run at different addresses in memory.

Relative Jumps

In the programming examples we've just seen, the destination of all the jumps was very close to the jump instruction.

However, we still needed a two byte address. The Relative Jump instruction gets around this by having a single byte op code and a single byte **DISPLACEMENT** from the address of the jump instruction to which control is to be passed.

The displacement represents a value between -128 and +127. It is thus stored in Two's Complement Representation. The displacement is the 'distance' over which the jump is to be made from the jump instruction address. Thus a displacement of -12 would cause a jump to an instruction 12 bytes before the jump instruction. We can thus pass control to addresses up to 128 bytes before the jump and up to 127 bytes after it.

The mnemonic for this instruction is:

```
JR      cc,displacement
```

where 'cc' is one of the conditions, applicable to these Relative Jumps. The JP instructions, by the way, are called ABSOLUTE JUMPS. Unconditional Relative jumps, such as:

```
JR      displacement
```

are also possible. the value of the displacement causes a jump in the following fashion.

1. The CPU adds 2 to the value that the Program Counter has when it's pointing at the op code of the JR instruction. the result of this is to point PC at the address immediately following the displacement byte.
2. The address following the displacement byte is thus used as the base address for the jump. Look at the below to make things clearer.

```
INC     A       -3
JR      Z,      -2
        02      -1
LD      A,       0
        02      +1
LD      B,      +2
        04      +3
```

The byte immediately following the displacement is numbered 0; the next +1 and so on. You should be able to see from the above that the statement JR -2 isn't a terribly good idea, because

all that would happen is that the CPU would repeatedly executing the JR -2 instruction, thus giving a never ending loop!

If you're using an assembler, then the displacement bytes needed to perform a jump to a particular label are carried out automatically. However, if you're putting the programs together by 'hand', that is, using the tables in the back of the book, then read on.

The negative displacements can be entered into DATA statements to be poked into memory as negative numbers, thus saving us from having to work out the Two's Complement representation of the displacements. If you do have to work out what the Two's Complement is then a quick way is to POKE the negative number into memory then PEEK it back.

As a final example, look at this short program section.

```
        LD      A,0
LOOP:   INC     A
        JR      NZ,LOOP
```

The displacement here would be -3, or 253 in Two's Complement. The displacement byte in Two's Complement is thus '256-value' for the negative displacements.

The conditions offered by the relative jumps are fairly limited. They are:

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

So, if you want to make a jump based on the parity of a number, no matter how small the jump is, you'll have to make an absolute jump with JP.

The major advantage offered by the relative jump instructions is that they make no absolute references to addresses in the computer memory. All the addresses to which jumps are to be made are specified **RELATIVE** to the current position in memory. A few minutes' thought will show that this indicates that a program written using relative jumps only will

run **ANYWHERE** in memory without alterations. Listing 8.3 shows a modified version of Listing 8.1. Assemble it once, save it to tape, then load it to a few different RAM addresses. You should find that it will work at any address in memory that isn't used by the computer OS for anything.

```
        LD      HL,16384    33,0,64
        LD      BC,6144     1,0,24
LOOP1:  LD      (HL),255    54,255
        INC     HL          35
        DEC     BC          11
        LD      A,B         120
        OR      C           177
        JR      NZ,LOOP1    32,-8
        RET                 201
```

Listing 8.3 Demonstration of relative Jumps.

Programs written like this, so that they can be executed at any address in memory, are called **RELOCATABLE** programs.

Register Indirect Jumps

We touched briefly on these at the start of the Chapter. They are jumps where the address to be jumped to is stored in the HL, IX or IY register pairs. The legal instructions in this addressing mode are:

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

You cannot have conditional Register Indirect jumps. As a concrete example, the below instructions will cause a system reset by jumping to address 0. The RET instruction is, of course, rather redundant!

```
LD    HL,0
JP    (HL)
RET
```

FOR...NEXT loops in machine code

We've now seen the machine code equivalents of the BASIC IF...THEN GOTO and GOTO. In BASIC, we also have the ability to execute a sequence of commands a set number of times using the FOR...NEXT structure. To do this in machine code, we use jump instructions, as we'll now see. For example. let's convert the program:

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

into machine code. Well, registers are used to replace the I and C variables, as you might expect. The below routine will simulate the above BASIC program.

```
        LD      C,0         ; set 'C'=0
        LD      B,6         ; initialise 'I'
LOOP:   INC     C           ; C=C+1
        DEC     B
        JR      NZ,LOOP
```

You'll see a couple of points of interest here. The first is that, as is often the case, we count down to zero rather than up to a certain value. This is because it's easier to check for a register containing the value 0 than it is to check for a register holding another value. The second point is that the combination DEC B and JR NZ, does turn up quite often in loops, but in a 'hidden' form. It was decided by the chip designer that the Z80 should have an instruction that decremented a register and did the JR NZ instruction in the same instruction. This instruction is called DJNZ, which stands for Decrement B, and Jump if Not Zero. The full instruction is DJNZ displacement, and the displacement is calculated in the same way as it was for the relative jumps. It only works with the B register, and for this reason the BC register pair is often favoured for general counting jobs by programmers. We can now rewrite the above program as:

```
        LD      C,0
        LD      B,6
```

```
LOOP:   INC     C
        DJNZ    LOOP
```

There's only one problem with DJNZ; as it only works on 1 register it can only take care of loops needing 256 or fewer repetitions. There isn't a 16 bit DJNZ. 256 loops, you say? How come? The largest number you can fit into an 8 bit register is 255, so how can we get DJNZ to repeat something 256 times? The answer is that we simply load the B register before the DJNZ instruction with the value 0. After the first DJNZ, a value of 255 will be left in the B register, thus allowing 256 passes around the loop.

DJNZ instructions can, with care, be 'nested' to allow loops to be executed with DJNZ that need more than 256 passes. We do this by using the stack to temporarily store the contents of the B register. For example,

```
        LD      B,16
OUTLOOP: PUSH   BC
        LD      B,255
INLOOP: ......
        ......
        DJNZ    INLOOP
        POP     BC
        DJNZ    OUTLOOP
```

We could always use a 16 bit register pair and decrement this, but that would require us to use A to check whether or not the register pair is holding 0 or not. This isn't necessary if we use the stack like the example above.

We could, of course, add a 'STEP' to our machine code FOR...NEXT loop in the below fashion.

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

This is the same as FOR I=0 TO 100 STEP 2:LET C=C+2:NEXT I. The STEP 2 is introduced by inserting an

extra DEC B command in addition to the one that's in the DJNZ instruction, and we increment C by two by simply adding an extra INC C instruction. The C register will count up 0,2,4.... and the B register will count DOWN 100,98,96... One problem that could arise is that if you were, in this particular example with two DEC B instructions, to put an odd number in B to start with then the B register would never hold the value 0; it would get as far as 1, be decremented twice, then assume the value 255! You would thus have a never ending loop. Watch out for this!

Table 8.1 shows the loop and jump instructions with the times they take. No flags are affected by these operations.

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

Table 8.1 Jumps and Loops

Where two times are given, the first is the time taken for the instruction to be executed when the condition is not met, and the second is the time taken for the instruction to be executed when the condition is met. A further point to note is that JR instructions are on the whole a little slower than JP instructions. This is due to the CPU having to calculate the address to be jumped to from the displacement byte given. Also, JP cc instructions take the same time to execute whether or not the conditions are met.

CALL and RETURN

In BASIC, we had the instruction GOSUB and RETURN that gave us the ability to use subroutines. A subroutine, you may remember, is a block of instructions that is stored once in memory but that can be called and used as often as you like in the 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 USR instruction from BASIC, we are making a subroutine call from the BASIC interpreter to our machine code. The RET instruction with which we have to follow our machine code if we want to get back to BASIC is the machine code equivalent of the RETURN instruction in BASIC.

In machine code, the

```
CALL nn
```

instruction calls a subroutine at address 'nn'. The instruction is a three byte instruction; a single byte op code and a two byte address. The address should be stored in memory with the low byte of the address stored first. Any piece of code that is called as a subroutine by a CALL instruction must end in a RET instruction. Once the RET is executed, the CPU, rather cleverly, goes back to continue the program from the instruction immediately following the CALL nn instruction. How does the CPU know where to go back to? After all, there might be many CALL instructions in the program.

Well, the stack is used, and this is the main role of the stack in the Spectrum 128. When the CALL is made, the CPU finds the address of the instruction immediately following the CALL instruction and puts it on to the stack. The RET instruction, when executed, gets the last two bytes off of the stack and uses them as the 'return' address; i.e. the address to carry on from after the CALL instruction. The CPU will then jump back to the return address.

Within the 'body' of the subroutine, therefore, it's vital that all the PUSHes on the stack are matched by POPs. If you issue a RET instruction with a number on the stack that isn't the

correct return address, then the CPU will jump to the wrong address! There are some programming techniques, that should be practised only when you gain some experience, in which the return address on the stack is actually changed so that the CPU treats the RET instruction as a kind of JP instruction, with the jump address held in the stack. However, let's first look at the normal behaviour of CALL and RET.

```
40000   CALL 41000 ----->41000 INC A
40003   INC A <----------41001 RET
```

Here, the RET instruction passes the control of the CPU back to the instruction that immediately follows the CALL, in this case the INC A at address 40003. This is the usual behaviour of the CALL and RET instruction.

If we now introduce a PUSH or POP then we can alter this situation, as shown below.

```
40000   CALL 41000---->41000  PUSH  AF
        INC  A <-------------  INC   A
                               POP   AF
                               RET
```

The PUSH at address 41000 is balanced by the following POP instruction, thus the number left on the stack for the RET to pick up will be the correct return address.

With unbalanced PUSHes or POPs though, the situation is very different.

```
40000   CALL 41000---->41000  LD    BC  ,0000
        RET                   PUSH  BC
00000 <-------------------------RET
```

The address 00000 is pushed on to the stack in the subroutine but isn't removed. So, when the RET instruction is executed zero is taken as the return address and so execution starts at address 0. This is the 'system reset' address in the Spectrum and so you'll lose everything!

If, however, we were to put the address of a routine of our own writing on the stack then we could use the RET as a kind of jump. For example:

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

The POP instruction removes the original address from the stack, and we then replace it with the value 42000, which the CPU will think is the return address when the RET is executed. The 'jump' will then be carried out to address 42000.

All this is very useful, but it's not the real use of CALL, which is to execute a subroutine then return to continue executing the program instructions immediately after the CALL instruction. You will note that the address in a subroutine call is a full, 16 bit address. There are no relative subroutine calls. Programs written with a subroutine calls in them are therefore not relocatable, unless the addresses to which the calls are made remain the same and hold the same information. Usually this is only true if calls to ROM routines are made.

Saving Registers

You will occasionally want to call subroutines but keep the contents of the registers intact for later use. This is commonly done by PUSHing the registers on to the stack and POPping them back after the subroutine has been executed. This is called **PRESERVING** the CPU registers, or, if you're into impressive sounding phrases, 'preserving the CPU environment'.

Always try and put the subroutine definitions- the piece of code that's to be run every time you make a subroutine call- into a place in memory where they won't be executed by the CPU without them being called. If this happens, it can lead to some puzzling problems. Subroutine calls are slower than just executing the code, but they obviously save memory. If time is absolutely crucial but you've got plenty of memory to play with, then it might be worth repeating blocks of code.

Conditional Subroutine Calls

In BASIC we can use IF...THEN GOSUB. Well, we have a similar sort of thing in machine code. This is of the form:

```
CALL cc,address
```

where cc is a condition. The CALL will only be made if the condition is true. The conditions that can be used are as follows:

```
CALL C,nn       call if carry true
CALL NC,nn      call if carry false
CALL Z,nn       call if result zero
CALL NZ,nn      call if result not zero
CALL PE,nn      call if parity even
CALL PO,nn      call if parity odd
CALL M,nn       call if result negative
CALL P,nn       call if result positive
```

No flags are affected by the CALL instruction. A typical use of the conditional CALL instructions might be:

```
LD      A,(choice)
CP      1
CALL    Z,option1
CP      2
CALL    Z,option2
```

and so on. 'Choice' could be the result of a key press. If it's '1' then the subroutine at 'option 1' is executed. If '2' then the code at 'option2' is run. One thing to note is that it might be useful to preserve the value of A on entering the subroutine and restore it on leaving the subroutine. If this isn't done then on return from executing one option the CPU might go off and try to execute a second option.

We can also use conditional RET instructions, which only cause a return from a subroutine when a particular condition is true. The conditions that are available are the same as for the conditional CALL statement. Table 9.2 shows the CALL and RET instructions. Again, where two times are given the shorter is the time needed when the condition is not met.

Mnemonic		Bytes	Time Taken	Effect on Flags CZP/VSNH	
CALL	address	3	17	---	---
CALL	cc,address	3	10/17	---	---
RET		1	10	---	---
RET	cc	1	5/11	---	---

Table 8.2 CALL and RET

Restarts

There are some rather special instructions in the Z80 instruction set called Restarts. These can be looked at as 1 byte subroutine calls. They are faster than CALLs, but they are to fixed points in memory. In the Spectrum, all these locations are used by the Spectrum 128/Plus 2 ROM, so the Restarts were used by the ROM programmer to call certain commonly used routines in the first 100 bytes of ROM. The addresses to which the Restart, or RST, instructions will call are usually written in hexadecimal, and they are &00, &08, &10, &18, &20, &28, &30, &38. The uses of some of these will be examined when we look at ROM routines at the end of the book. RST routines usually end in a RET instruction.

RST &00 will cause a jump to address 0 and reset the computer. RST &08 is used to generate an error message by the BASIC Interpreter. Try the below program:

```
RST     &08
DEFB    11
RET
```

The program will generate an error message. Try other numbers instead of 11 in the DEFB staement. You will find that the error generated differs as different numbers are used.

Block Operations

These operations operate on several bytes instead of the more usual one byte, and also they are made up of several Z80

instructions rolled into one. This makes them faster than a program written using the corresponding sequence of instructions.

The CPI Instruction

The simplest of these is:

```
CPI
```

and this stands for ComPare and Increment. The instructions compares the current contents of the A register to the contents of the byte whose address is in the HL register pair. The HL register pair is then automatically incremented. CPI thus consists of the two instructions:

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

A use for this command is to search a block of memory for the occurrence of a particular byte. Listing 8.4 shows an example of the use of this program, to search through memory from location 1000 onwards for the first occurrence of the byte stored in address 41000. The address at which this was found is then stored in addresses 41001 and 41002 from where it can be PEEKed out. The only other point to note is that the displacement for the relative jump is -4, due to the CPI instruction having a two byte op code.

```
            ORG     40000
            LD      HL,1000
            LD      A,(41000)
SEARCH:     CPI
            JR      NZ,SEARCH
            DEC     HL
            LD      (41001),HL
            RET
```

Listing 8.4 CPI Demonstration

To see the program work, POKE a value into address 41000, RANDOMISE USR 40000, then PRINT PEEK(41001)+ 256*PEEK (41002). If you like, modify the program so that the

contents of HL after the DEC instruction (this is necessary because the CPI increments HL immediately after the comparison) are transferred to the BC register pair. The program can then be executed by using PRINT USR 40000, and the address will be printed to the screen. The DEC HL instruction used doesn't affect the flags, so the Z flag will only be set to zero if the comparison operation was successful.

The CPD Instruction

A similar instruction, CPD, performs a similar job but this time the HL register pair is Decremented after the comparison operation. With both CPI and CPD, though, it's not just the HL register pair that's affected. The BC register pair is decremented by both instructions. The Increment or Decrement mentioned in the name of the instruction only applies to the HL register pair. The fact that BC is decremented allows us to search through a block of memory of a given length for a particular byte. Listing 9.5 is similar in function to listing 9.4, but now only searches 255 bytes of memory after address 1000.

```
            ORG     40000
            LD      BC,255
            LD      A,(41000)
            LD      HL,1000
SEARCH:     CPI
            JR      Z,FINISH
            INC     C
            DEC     C
            JR      NZ,SEARCH
FINISH:     INC     HL
            PUSH    HL
            POP     BC
            RET
```

Listing 8.5 CPD demonstration

Because the 16 bit decrement 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 to see if the C register is zero. The DEC instruction will set the Z flag if the C register was zero when we did the INC instruction. The program returns the value 255 if the byte isn't found, or the address if the byte is found. The routine is called with PRINT USR 40000 after poking the byte to be searched for into address 40000.

CPIR and CPDR

These two instructions are really powerful; they are two byte instructions that are the equivalent of a CPI or CPD instruction with a built in loop.

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, HL holds the start address of the block to be searched and BC holds the number of bytes to be searched. The instruction will terminate for one of two reasons:

1. A match has been found.
2. The end of the block has been reached.

Thus, after a CPIR or CPDR instruction we have to test to see which condition terminated the instruction. This isn't such a difficult task as it sounds. Simply remember that if the block has been totally searched then BC will hold zero. Therefore, we just have to check for this condition. This is shown below:

```
LD      HL,1000
LD      BC,1000
LD      A,255
CPIR
LD      A,B
OR      C
JR      Z,END-OF-BLOCK
```

The label 'END-OF-BLOCK' is jumped to if the BC register pair contains zero when the CPIR instruction finishes. Otherwise, the termination was due to the instruction finding the desired byte. These instructions are fairly time consuming to execute, but they are faster than using the individual compare and jumps.

Block Moves

Occasionally we may want to move whole chunks of memory around. One way to do this would be to use a simple program such as:

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

Here we transfer 200 bytes from address 40000 onwards to address 42000 onwards. This is a copy operation. The bytes will still be present at the two hundred locations from 40000 onwards. The HL register pair points to the byte that we're copying, and DE to the place in memory to which we wish to copy the bytes. The B register is used to count the number of bytes to be copied. This program will work, but is rather inefficient, as there are instructions in the Z80 instruction set that do this sort of thing automatically. The first of these instructions is called LDI. We can rewrite the above program as:

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

LDI is similar to CPI. BC is decremented, and the contents of the byte addressed by the HL register is transferred to the address in HL. In addition, HL and DE are Incremented. DE is thus called the DESTINATION register and HL the SOURCE register. LDD does a similar job, but here the DE and HL registers are decremented instead of incremented. A further increase in the efficiency of the above program can be obtained if you note the fact that after LDI or LDD the P/V flag is set if the BC register DOES NOT contain zero. We can thus use this flag to decide whether a loop around is needed or not.

To make matters easier still, we have two more instructions called LDIR and LDDR which are automatically looping versions of LDI and LDD respectively. These use the P/V flag mentioned above. For example, the short routine below will transfer 2000 byte from ROM to screen memory.

```
ORG     40000
LD      HL,0000
LD      DE,16384
LD      BC,2000
LDIR
RET
```

Listing 8.6 LDIR demonstration.

Table 8.3 shows the timings and flag effects of the block instructions.

Mnemonic	Bytes	Time Taken	Effect on Flags CZP/VSNH
LDI	2	16	--# -00
LDD	2	16	--# -00
LDIR	2	21/16	--0 -00
LDDR	2	21/16	--0 -00
CPI	2	16	-## #1#
CPD	2	16	-## #1#
CPIR	2	21/16	-## #1#
CPDR	2	21/16	-## #1#

\# indicates flag changed
1 indicates flag set to 1
0 indicates flag set to 0
\- indicates flag unaffected

Table 8.3 Block Operations.

For repeat instructions, the time shown is for the transfer of a single byte. Shorter times are for the termination of the instruction.

Chapter 9
Ins and Outs and Odds and Ends

In this Chapter, we'll look at the few remaining Z80 instructions that we haven't covered. These are the input and output instructions and a few instructions that don't really fit into any other category.

Input and Output Instructions

As well as being able to communicate with the RAM and the ROM, the CPU can also read information from and write information to a variety of addresses called IN/OUT or I/O addresses. There are 65536 of these available to the CPU, but not all of them are available in the Spectrum due to the way in which the Spectrum hardware is arranged. Amongst other things, they allow the CPU to communicate with the keyboard, PSG, Microdrives or joysticks, amongst other things. As a beginner, you'll probably only use the I/O commands to use the Programmable Sound Generator. However, as you gain experience you'll probably want to use some of the other I/O devices directly with In and OUT instructions. All I'll do here is indicate the instructions that are available, and look at a few of the more important I/O addresses.

The simplest I/O instructions are:

```
IN      A,(n)
OUT     (n),A
```

The input instruction, IN, reads a single byte from the I/O device whose 8 bit address is 'n' and puts the value in the Accumulator. OUT sends a byte from the accumulator to the I/O device with address 'n'. However, these commands aren't often used; it's more usual to specify a 16 bit I/O address using the BC register pair to hold the address. The instructions that this mode of operation uses are:

```
OUT     (C),A
IN      A,(C)
```

where BC holds the I/O address. For example,

```
LD      BC,49149
LD      A,25
OUT     (C),A
```

will send the value 25 to I/O address 49149. On the whole, unless you are adding extra equipment to the Spectrum, you won't need to use I/O instructions very often, as the ROM can usually do the job for you. However, for those of you who may be adding extra pieces of equipment, here is a description of I/O addresses used by the Spectrum. Addresses that use the address lines A0 to A4 are used by the Spectrum for its own work, and no more than 1 of these lines should be taken to 0 by an I/O command at the same time. If this were to happen, the CPU would try and read from or write to two or more devices at once! Not a very clever thing to do. Here is a list of what these lines are used for:

A0	keyboard, cassette, loudspeaker, border
A1	memory paging, serial interface, midi, sound.
A2	ZX Printer
A3,A4	Interface 1, for microdrives

As you'll see from other Chapters in this book, these address lines are used in conjunction with others to provide an address. It is possible to crash the computer if you read and write certain

I/O addresses, particularly those to do with the paging of memory. So, you have been warned.

IN instructions will affect the status of the S, Z and P/V flags when they load data into A. Instructions are available that do similar jobs to LDIR and LDDR but with I/O addresses rather than memory locations. However, these aren't much use on the Spectrum.

Odds and Ends

I want to use the rest of this short Chapter to look at instructions that won't fit in any where else in the book! Hence the 'Odds and Ends' part of the title. The first instruction that I want to examine is really useful, despite the fact that it does absolutely 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 very useful, for wasting time to slowing things down or for deleting instructions if you're 'hand assembling' programs. We can replace any instructions that we don't want by the op code for NOP, which is 00. Why bother with this? Well, if we've got any jumps in the program just deleting instructions would result in the jump addresses or displacements being incorrect. So, if we replace each byte of the offending instruction with a zero byte the displacements and addresses will still be the same because we haven't altered the number of bytes in the program.

RRD and RLD

We saw some time ago how we could use shift instructions to alter the value held in a register. We also saw the rotate instructions. All of these worked on single bits within a byte. Well, here are two similar commands that work on nibbles within bytes. The instructions only work in the Register Indirect addressing mode. Note how the A register is also involved in these operations.

RLD

In general terms this can be represented as:

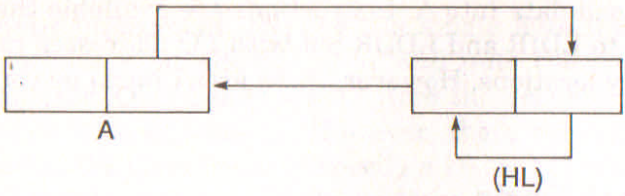


Figure 9.1 RLD

As can be seen, the operation involves a byte addressed by the HL register pair and the contents of the A register. For a particular example, let's consider:

A=0010 1100 (HL)=1010 0010

After the RLD instruction we've got:

A=0010 1010 (HL)=0010 1100

RRD

In general terms, this can be represented by:

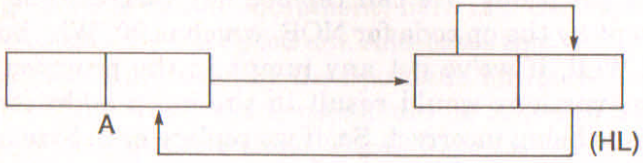


Figure 9.2 RRD

For a particular situation, we can consider the following:

A=1010 0001 (HL)=0010 0110

After the RRD command we end up with:

A=1010 0110 (HL)=0001 0010

These commands are really only of great use if you're using BCD operations.

HALT

This instruction causes the CPU to stop until an interrupt comes along. This happens every fiftieth of a second on the Spectrum, so any HALT state won't last long.

NEG

This instruction negates the contents of the Accumulator by carrying out an automatic Two's Complement on A. This is simply the process of complementing and incrementing the A register. Thus the two instructions:

```
LD      A,3
NEG
```

will leave the Two's Complement representation of -3 in A. 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 P/V is set to 1. Otherwise C=0 and Z and S have values depending on the result of the operation.

Well, that covers all the Z80 instructions that you'll need for the vast majority of all programming in machine language on the Spectrum machine. For the rest of the book, we'll see how we can apply what we've learnt to the machine itself.

Chapter 10 Interrupts on the Spectrum

While I was writing this book, a variety of things dragged me away from a red hot wordprocessor. They included phone calls, people at the door, food, sleep, etc. However, before leaving the machine I saved what I had written, made a note of where I was, then, after dealing with the interruption, came back, picked up from where I left off and carried on until the next interruption. These 'events' occur on a very frequent basis in computers. They are situations which require the immediate attention of the CPU, irrespective of what the latter is doing. The CPU saves what it's doing on the stack and then jumps off to a special routine in the computer memory that allows it to deal with the problem that has arisen. The event that triggered all this activity is called an **INTERRUPT**, and the piece of machine code that the CPU executes in response to an Interrupt is called the **Interrupt Service Routine**, or ISR. Once the ISR has been run, the CPU gets from the stack the address of the instruction that it was to execute before the Interrupt occurred, and goes back to deal with it.

There are two types of Interrupt that the Z80 can respond to. These are called **MASKABLE** and **NON MASKABLE** interrupts. Maskable interrupts can be 'ignored' by the CPU when they occur. In the above example, I could just let the

phone ring, for example, and take no action with regard to it. The Non Maskable Interrupt, or **NMI**, is always acted upon by the CPU. They cannot be ignored. Both signals are sent to the computer using pins on the CPU chip; the NMI pin is called NMI, and the Maskable Interrupt pin is called **IRQ**, short for **I**nterrupt **ReQ**uest.

EI and DI

These two instructions allow us to program the CPU to respond or not respond to an IRQ (Maskable Interrupt). If an EI (Enable Interrupt) instruction is executed, then the Z80 will respond to maskable interrupts. If DI (Disable Interrupts) is executed, then the Z80 will not respond to any maskable Interrupts. You'll often hear programmers say that they are 'turning the interrupts on' or 'turning the interrupts off' using these commands. On the Spectrum, you should ALWAYS make sure that the interrupts are 'on' or **ENABLED** by using an EI command before you return to BASIC. This is the normal way in which the Spectrum operates, and failure to enable the interrupts before a return to BASIC will cause the machine to 'freeze'.

The Spectrum and Interrupts

Interrupts allow a CPU to run two programs 'at once'. A main program, called the FOREGROUND PROGRAM, is executed most of the time, and this is occasionally interrupted so as to cause the CPU to run a second, shorter, program. This is how the Spectrum uses interrupts. A maskable interrupt is produced every 50th of a second by the Uncommitted Logic Array (ULA). When the CPU receives this interrupt request, it executes a routine at address &0038 in ROM. This address, you will remember, is one of the Restart addresses, and is the start of a routine that reads the keyboard of the Spectrum and updates the value of a three byte system variable called FRAMES (addresses 23672 to 23674). The latter thus form a counter that is incremented every 20 milliseconds (every

1/50th of a second). The low byte is first, and you can use this counter for timing applications.

The Spectrum executes the BASIC program a statement at a time, and after each statement a HALT instruction is executed by the BASIC ROM. This causes the CPU to 'mark time' until an interrupt is received. Thus the running of the BASIC program is also dependant on Interrupts being enabled. You can probably see, therefore, that disabling interrupts then returning to BASIC would cause severe problems, to say the least; the BASIC program wouldn't be run and the keyboard wouldn't be read!

However, interrupts can be disabled whilst we're in machine code, and the Spectrum ROM does this quite often. If you think about it, a task that's being interrupted occasionally will run more slowly than if it were not interrupted. Also, there's no way in which we can tell precisely when an interrupt is going to occur; it might happen at a time when the timing of a particular operation is absolutely crucial! So, in the ROM, when an application is very dependant upon timings being accurate the Interrupts are temporarily disabled. So, when a BEEP is executed, or we're carrying out tape saving and loading operations, the interrupts are disabled. Interrupts are also disabled by the Interface 1 device so as to allow Microdrive operations to be uninterrupted. Of course, the Spectrum Operating System will turn on the Interrupts after it's completed these routines where timing is important. Whilst interrupts are turned off, the keyboard will not be read and the FRAMES counter will not be updated. Thus extensive use of the DI command could cause FRAMES to lose time.

Interrupt Modes

As the Spectrum normally operates, an interrupt always causes the routine at address &38 (decimal 56) to be executed. This isn't much use to us if we want to 'commandeer' the interrupts of the Spectrum for our own use. The Z80, however, can be programmed to respond to interrupts in different ways

by altering what is called the INTERRUPT MODE in which the CPU operates. In the 128 the Z80 is usually set to operate in Interrupt Mode 1, which simply tells the CPU to execute the instructions found at address &38 whenever an interrupt occurs and interrupts are enabled. This is set by the instruction:

```
IM    1
```

and this instruction is executed by the ROM whenever we reset the machine or type in the command NEW. There are three interrupt modes altogether, modes 0,1 and 2. The important one for us in actually using Spectrum Interrupts is Mode 2, which is set by the:

```
IM    2
```

instruction. This causes the CPU to execute the instructions found at an address in memory that we specify. Thus we can use this interrupt mode to execute some of our own machine code every fiftieth of a second rather than the normal routine at address 56. The problem is, how do we tell the Z80 where our interrupt routine is?

Interrupt Vectors

When we use Interrupt Mode 2, we use another CPU register called the I register. This is not the 'I' of the 'IX' or 'IY' register pair, it is a register that is used by the CPU to work out where in memory the Interrupt Service Routine is.

In IM 2 the Z80 expects to find the address of the ISR in a two byte location in memory called an **INTERRUPT VECTOR.** This is NOT the start of the routine, but is the address of the start of the Interrupt Service routine. A vector is a bit like the index of a book; it doesn't hold the information contained in the book but tells you exactly where to find it in the book. So, if we've got a vector holding the address of the ISR, there just remains the problem of telling the CPU about the whereabouts of the vector in the memory. This is where the I register comes in; the I register is used to hold the high byte of the address of the vector. This now leaves us with the

low byte. The CPU works out the full address of the vector by using the I register contents as the high byte and the value on the data bus at the instant of the interrupt. Now, in the Spectrum this is usually 255, so that all vector addresses that the Z80 can use in the Spectrum will be &XXFF, where 'XX' is the hexadecimal value of the I register. So, if we put a value of &FD into the I register the vector should be at locations &FDFF and &FE00; the I register and data bus values together specify the first byte of the vector. The address in the vector is stored in the usual Z80 low byte first fashion. Thus, the byte in address &FDFF would be the low byte of the address of the Interrupt Service Routine and &FE00 would hold the high byte of the address of the ISR. However, a few complications can arise. The first of these is that certain peripherals, when connected to the Spectrum, can cause the data bus to hold other values than 255 when it's not carrying information around. Thus, the data bus might be carrying the value 254. The low byte of the address of the Interrupt Vector thus might not be &FF if these peripherals are connected. The simplest way around this problem is to fill 256 bytes of memory with the same value, so that no matter what the value on the data bus when the interrupt occurred we'd always get the right address for the ISR. For example, if our I register was set to hold the value &80, the normal situation would be for the ISR address to be picked up from address &80FF and &8100. However, if the data bus were to be holding &FE, the ISR address would need to be in addresses &80FE and &80FF for the program to work correctly. Now, if we filled all the memory between &8000 and &8100 with the same byte, say &BF, then no matter what value ended up on the data bus from peripherals the ISR address would always be read as address &BFBF. You could then put the ISR routine at address &BFBF in memory.

The second problem is that some values of the I register will cause problems if used as vector addresses. The reason for this is that the I register is also used by the computer to display the contents of screen RAM. There can be competition

between it's use in the video function of the computer and its 'proper' use in Interrupt Handling. This is called **CONTENTION**, and the values mainly affected are values of &40 to &7F in the I register. These values will cause problems whether the computer is operated in 128 or 48 mode. If we're using 128 mode, and we're using RAM pages 4-7 paged into the last 16k of memory space, then contention problems can arise if we use I register values of &C0 to &FF. Thus, to be perfectly safe we should use I register values of &80 to &BF, thus allowing us to use addresses between &8000 and &BFFF as vectors. To simplify matters, the example programs given below will use the machine running in 48 mode. In addition, it's a good idea when using interrupts to put the ISR into addresses above 32767. This is because below this value the ULA is involved in the display of screen information, and can get priority, if it needs it, over the CPU which might be getting machine code instructions from addresses elsewhere in this 'lower' RAM. The time that the CPU takes to execute instructions can thus vary. An example of this can be found in Chapter 13.

Simple Interrupts

To use Interrupt Mode 2, the Z80 has to be put into this mode of operation. This is done in the following way:

```
DI
LD      A,vector-hi
LD      I,A
IM      2
EI
RET
```

The interrupts MUST be disabled before we do this; otherwise if one was to occur half way through the job of switching modes and setting up the vector we'd have problems. 'vector-hi' is the high address of the vector. It's important that when we set the vector up and re-enable the interrupts, the vector should hold the ISR address AND the ISR routine should be in place in memory. The IM 2 instruction just sets up the Interrupt Mode. Then we simply re-enable the interrupts and finish.

To set the Spectrum back to its 'normal' way of handling interrupts, we can use a piece of code such as:

```
DI
LD    A,&3F
LD    I,A
IM    1
EI
RET
```

The I register normally contains the value &3F (decimal 63), and so we set the I register back to this and then set Interrupt Mode 1 up again. Again, note that interrupts are disabled while we're doing this operation.

As to the program that you execute whilst you're running interrupts, it should save all registers that are to be used by the ISR on the stack as the first thing that it does and then recover the register contents from the stack before it finishes. This is because the routine that was interrupted will want to carry on from where it was; if it can't find the expected values in the registers then a crash can be expected. Listing 11.1 does nothing visible, but has actually redirected the Spectrum interrupts to our own routine.

```
        ORG     40000
change: LD      HL,isr
        LD      (&FDFF),HL
        EI
        LD      A,&FD
        LD      I,A
        IM      2
        EI
        RET
isr:    DI
        RST     &38
        EI
        RET
off:    DI
        LD      A,63
        LD      I,A
        IM      1
        EI
        RET
```

Listing 10.1 Interrupt Demonstration

The vector in use is at address &FDFF and &FE00, and this is loaded with the ISR address from the HL register at 'change'. This also sets the Interrupt Mode to 2 and sets I up to use the ISR vector. The actual ISR simply does the normal function performed by interrupts in mode 1 on the Spectrum by calling the routine at address decimal 56. Interrupts are disabled in the ISR so that a second interrupt cannot occur whilst we're dealing with the first. This is particularly important for longer ISRs than we'll use in this book. To set the new interrupt mode into operation, PRINT USR(40000). The machine should go on as usual. To turn it off, call the 'off' routine.

Listing 10.2 shows a program that increments a 16 bit memory location at each interrupt. This is similar to the FRAMES system variable. Listing 10.2 will run in 128 or 48 mode.

```
        ORG     40000
        LD      HL,isr
        LD      (&AFFF),HL
        DI
        LD      A,&AF
        LD      I,A
        IM      2
        EI
        RET
isr:    DI
        PUSH    AF
        PUSH    HL
        LD      HL,(50000)
        INC     HL
        LD      (50000),HL
        POP     HL
        POP     AF
        EI
        JP      &38
```

Listing 10.2 Counter

Here, the interrupt mode is set up and the Interrupt Vector is set up by a PRINT USR(40000) command. This will then cause the routine at 'isr' to be executed every 1/50th of a second. This will increment the contents of address 50000 and 50001. The PUSH instructions simply save the AF and HL register pairs. The last instruction of 'isr' simply jumps back to the routine at address &38 that is normally executed by the Spectrum Interrupts. Repeatedly PEEKing the contents of 50000 and 50001 will show that the routine is working.

Of course, the longer your ISR is then the 'slower' the updating of the keyboard and FRAMES will be due to the increased length of time between subsequent calls to address &38. If you are using FRAMES, it's a good idea, from BASIC, to read the three byte value twice and use the higher valued one as the correct response; this gets around a problem that can occasionally turn up in that you read the value of FRAMES as it's being updated.

That is as far as we'll go in looking at interrupts in this beginners book. There are other things that you can achieve with interrupt driven routines, but they require some programming experience. However, we'll leave it here.

Chapter 11
48 and 128 Modes of Operation

The 128/Plus 2 is equipped with a total of 128k bytes of memory, all crammed in to the 64k memory map of the Z80 CPU. As we've already indicated, this is done by the technique of PAGING memory, which allows different blocks of memory to occupy the same space in the memory maps, though not at the same time. Whether or not a given block of memory is 'paged in' at any given time depends upon whether it has been selected by a program or not; clearly, two blocks of memory in the same part of the address map cannot be paged in at the same time.

There are two separate ROMs, the Editor ROM and the Spectrum BASIC ROM, and these both occupy the lower 16k of memory space. When you're editing a program, the Editor ROM is paged in, but the rest of the time the BASIC ROM is paged in, thus allowing us access to the BASIC ROM routines in our machine code programs. If we want to get access to any of the routines in the Editor ROM, and there are a couple of useful ones, then we have to call them at special 'entry points' in the BASIC ROM. This is beyond the scope of this book, but suffice to say that the code at the 'entry points' ensures that the Editor ROM is paged in and the desired routine executed, and then that the BASIC ROM is repaged before finishing.

This, by the way, is similar to the approach taken by Sinclair when the Microdrives arrived on the scene; rather than re-write the ROM, a 'piggy back' system was invented which allowed the new ROM which controlled the microdrives to be called up only when needed.

There are some obvious differences between 48 and 128 modes on the computer; the single key entry of BASIC keywords is absent in 128 mode, PLAY won't work in 48 mode, and, in 128 mode, you've got access to the Paged RAM. 128 mode uses the paged RAM in the following way:

1. 1 page of RAM is used by the screen editor.
2. The rest of the paged memory can be used as a storage medium called a 'silicon disc'. The traditional disc drive is a very fast storage medium, allowing us to save and load bytes much more quickly than we can with tape or Microdrives. The 'silicon disc'. isn't a permanent storage medium, though. As soon as the power is removed anything that we've stored in the Paged RAM will be lost, just like the contents of the rest of the RAM in the computer. Data, including BASIC programs, can be stored in the paged RAM using the normal tape filing system commands but with a "!" in the instruction. For example, SAVE ! "fred" will save the current BASIC program as a 'file' called 'fred' into the paged memory. LOAD ! "fred" will recover the 'file' into program memory.

We machine code programmers can make similar use of the paged RAM, as we'll soon see.

New System Variables

On the old Spectrum, the area of memory between address 23296 and 23552 was used as a printer buffer by the now defunct 'ZX Printer'. In addition, programmers often used to store machine code routines in this block of RAM. In 128 mode, though, this whole area is taken up with new system

variables. For this reason, programs written on the old Spectrum might not run in 128 mode if they used this area of memory for storage of data or machine code, and you'll have to go into 48 mode and convince your 128/Plus 2 that it's an 'old' Spectrum!

The variables in this area of RAM aren't as useful to the beginner as those in the main System Variable block. They are used for the 'Silicon Disc' storage system, the add on keypad and the paging in and out of ROM.

Paging Memory

There's no real benefit to be obtained for the beginner by paging the Editor ROM in and out. However, we can make good use of the extra RAM, using it for storage of data or our own machine code programs. Such programs can then be 'down loaded' into the lower reaches of memory, executed, then overwritten by new programs, thus allowing very large programs to be stored in memory. Screen images can also be stored, allowing us to have several full screen images in memory at once; we can then switch between them at leisure, giving us an electronic 'slide projector'. All we need to do is alter a single I/O address, at address 32765. (Figure 11.1)

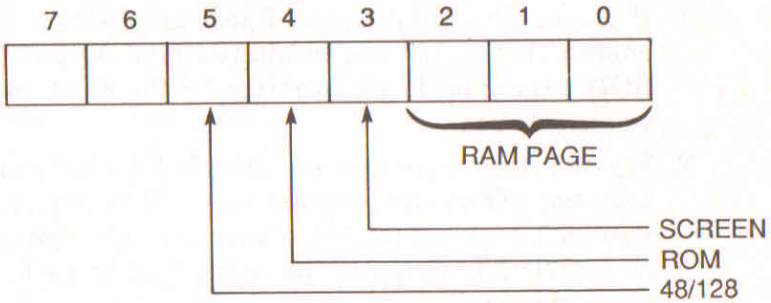


Figure 11.1 Memory Paging Address

BITS D0 TO D2 are used to specify which of the pages of RAM, 0 to 7, are paged into the top 16k of memory. Normally,

Page 0 is in place, with Page 7 being used for the screen editor. As we'll soon see, we can't just go around paging memory in and out as we please, we have to take care.

BIT D3 determines which 'screen' memory is used. Normally it's set to 0.

BIT D4. This is the ROM select bit. When set to 0, the editor ROM is selected. When set to 1, the BASIC ROM is selected. However, ROM paging isn't quite as simple as just setting this bit to either 1 or 0.

BIT D5. If set to 1, this will cause the 128/Plus 2 to become a 48k Spectrum!

Using the I/O Address

The paging byte is accessed by an OUT instruction using the BC register pair to hold the address of the I/O port. For example:

```
LD      BC,32765
LD      A,32
OUT     (C),A
RET
```

will set bit 5 and so turn the Plus 2 into a Spectrum.

When we're paging RAM, there are a couple of points to watch out for.

1. If you're intending to use Spectrum ROM routines, ensure that, after you've altered bits D0 to D2, the ROM paging bit is set correctly for the ROM you'll be wanting to use.
2. Try and make sure that you DON'T page out your own program! Thus any program that will be paging RAM shouldn't be in the top 16k of memory unless you've got an identical program in the same place in each of the pages to be used.
3. If you've paged in RAM other than page 0, remember that the stack will have been paged OUT. When paging RAM, make sure that any PUSHes, POPs, CALLs or RETs are done with the stack paged in.

4. Even if you don't use the stack, remember that interrupts are occurring while you're running your machine code programs, and the interrupts WILL try to use the stack. So, when paging memory always disable interrupts at the start of such an operation and enable them at the end.

To conclude this chapter, a couple of short programs to demonstrate the use of paged RAM to store screen images. Listing 11.1 transfers the contents of the screen RAM, starting at address 16384, to address 54000 in page 4 of RAM. The Attributes File (see Chapter 16) is not transferred.

```
ORG     30000
DI                      ;disable interrupts
LD      HL,16384
LD      DE,54000
LD      BC,32765        ; I/O address
LD      A,20            ; byte to send.
                        ; 16 = Spectrum ROM
                        ; 4  = Memory Page
OUT     (C),A
LD      BC,6144         ; number of bytes
LDIR                    ; move into paged RAM
LD      A,16            ; RAM page 0, Spectrum
                        ; ROM paged in.
LD      BC,32765
OUT     (C),A
EI
RET
```

Listing 11.1 Transfer screen image.

The reverse of this process, transferring the data from paged RAM into screen RAM, can be easily done by altering two of the instructions in the above from:

```
    LD      HL,16384
    LD      DE,54000
```

to

```
    LD      HL,54000
    LD      DE,16384
```

Note that the program is stored fairly low in memory so that it won't be accidentally 'paged out' when it's run.

Chapter 12
Sound

Sound on the Spectrum 128, when in 128 mode, is produced by the **Programmable Sound Generator**, a chip specially designed for this purpose. PSGs are used so that the CPU can be freed from the tedious job of sound production; all it needs to do is to send instructions to the PSG to tell it what sounds are needed, and the Sound Chip will do the rest. Clever, huh?

The PSG used in the 128 is called the **AY-3-8912**, (more often than not this is abbreviated to 8912) and as well as producing sounds is also responsible for providing the hardware needed for the MIDI/RS232 port and the add on keypad. As you can see, the 8912 is a very busy chip . . .

Before we go on to look at sound production, let's find out how we can actually hear the sounds produced! As you're probably aware, the 128 hasn't got an internal speaker. The sound output is 'mixed' with the video output to the TV set so that you can hear the sounds reproduced by the loudspeaker of the TV set. This is quite adequate, but the sound output is also available at the 'EAR' and 'MIC' sockets of the tape interface. This allows you to take the sound signal and feed it into an amplifier. Experiment with the sockets to find the one that gives the best quality sound output. Those of you who are interested in electronics should be able to make a simple

amplifier to use this facility; alternatively, you can pick up small battery powered amplifiers from shops specialising in electronics components for about £10.00 or so.

Whatever type of amplification that you use, the output from the Spectrum, though not terribly high, can drive an amplifier into 'distortion', where the sound begins to become 'rough' sounding and unpleasant. If this is the case, back off the volume control until the sound is clear again. Distortion can also be caused if you use a tape recorder as an amplifier, and leave both MIC and EAR plugs connected to the tape recorder and computer at the same time. This distortion is caused by feedback, and the cure is simple; disconnect one lead.

The MIC socket can also drive earphones, though not very well. I've used both the low impedance earphones intended for use with transistor radios and the high impedance 'hearing aid' style earpieces. The low impedance ones are marginally better. These can be useful when you are working in a noisy environment or when other members of the household might be bothered by your mysterious beeps and burps!

Software Sound Production

Although we've mentioned that the Spectrum 128 chip that can produce sounds, it's also possible to get the CPU to produce sounds via a suitable program. This isn't as efficient as when we are using the PSG, because the CPU cannot do anything else whilst it's producing the sound. However, if you're writing a program that is to run on a 48k Spectrum or Spectrum+ then you'll have to use this technique. It will also work on the 128 in 128 mode, though. The easiest way is to use the BASIC ROM sound routine, which can be called at address #03B5. There are no difficulties in using this outline, despite the fact that the ROMs are 'paged', as we learnt earlier in the book. This is because the BASIC ROM is paged in whenever we're running a program. The ROM routine is used in the following way; the DE register pair is set up with a value representing the duration of the sound, and the HL pair with a value

representing the pitch of the sound. These values are given by the equations:

```
DE   contents=F * T
HL   contents=(437500/F)-30
```

where F is the frequency of the tone that is required, in Hertz, or cycles per second and T is the duration of the sound you require in seconds. The frequency of a sound signal is the number of complete cycles of that sound in one second; Figure 12.1 shows this.

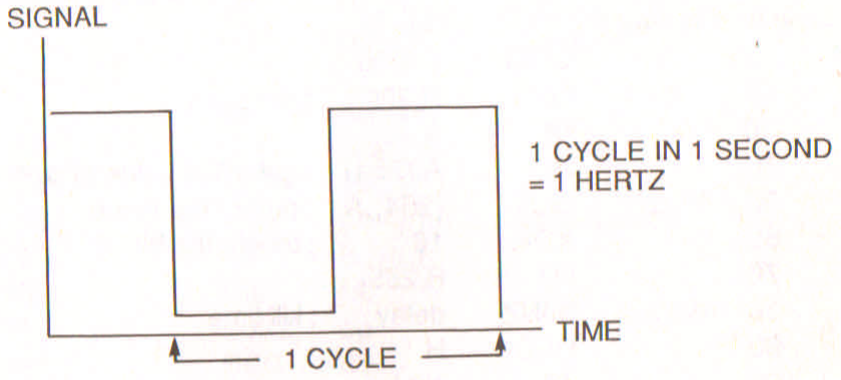


Figure 12.1

As a demonstration of the use of this ROM routine, try the below program.

```
LD     HL,1024
LD     DE,1200
CALL   #03B5
RET
```

The routine can be entered into any address and will work in either 48 or 128 mode. As was mentioned, this routine is very intensive of CPU power, not allowing it to do anything else while the sound is being produced. In addition, we've got no control of the volume of the sound produced except by altering the volume control of the amplifier or TV set! A more detailed view of this ROM routine can be found in *The Complete Spectrum Rom Disassembly* by Dr. Ian Logan & Dr. Frank O'Hara, also published by Melbourne House.

When the ROM routine makes the sound, all it is doing is repeatedly turning on and off a single bit of an I/O address – to be precise, I/O address 254. We can therefore write our own software sound routines using a similar technique. So, let's get on with it. The first thing to note is that port 254 is also responsible for the keyboard reading and border colour control. Therefore, we have to make sure that when we alter the port to turn the sound output on and off we mustn't alter any of the other bits of the port. Bit 3 is used for the MIC and EAR sockets. Listing 12.1 shows the simplest method of producing a tone by altering the value of this bit.

```
10              ORG     60000
20              LD      H,200     ; duration
30              DI
40              IN      A,(254)   ; get initial value of port
50      loop:   OUT     (254),A   ; output the value
60              XOR     16        ; toggle the bit
70              LD      B,255
80      delay:  DJNZ    delay     ; kill time
90              DEC     H
100             JR      NZ,loop
110             EI                ; enable interrupts
120             RET
```

Listing 12.1

Note how it's necessary to disable the interrupts of the machine before we start toggling the bit on and off. This is to ensure that the time taken in each loop is the same. If it weren't, then we'd have a sound produced of varying pitch. The pitch produced depends upon the value put in the B register within the loop, and the duration of the sound depends upon the pitch AND the value put into H. If you assemble this routine to run below address 32767 then the tone produced will be very 'rough' sounding. This is due to the ULA, responsible among other things for the screen handling, getting in the way. This effectively alters the rate at which instructions are fetched by the CPU from memory, and hence the speed at which machine code programs are run. Thus any

'time critical' routines should always be run at addresses above this value. Again, the routine will run in 48 mode as well as 128 mode.

As well as tone, you can also produce 'random' noise, as used in sound effects to produce something that's reminiscent of rain, explosions, gunfire, etc. This can be done by making the value put into B DIFFERENT on each cycle of the loop. Program 12.2 does this. Again, assemble it above 32767.

```
10          ORG   60000
20          LD    HL,0000    ; start of memory
30          LD    DE,#4000   ; 'duration'
40          DI
50          IN    A,(254)
60  lp:     OUT   (254),A
70          XOR   16
80          LD    (temp),A   ; store A value
90          LD    B,(HL)
100         INC   HL
110 dp:     DJNZ  dp
120         DEC   DE
130         LD    A,E        ; check DE for 0
140         OR    D
150         LD    A,(temp)   ; recover A
160         JR    NZ,lp
170         EI
180         RET
```

Listing 12.2

The program doesn't really produce random noise; as you can see, we're simply reading bytes from ROM and using them to provide a delay. Parts of the ROM filled with bytes of the same value will produce a tone of an even pitch, though there aren't that many such areas in the ROM and none are very long! Some programmers have even produced a reasonable facsimile of speech using this technique! However, that's rather beyond the scope of this book. In general, if we want to produce random noise we use the PSG. So, let's now leave methods of sound

production that are purely software based and look at the PSG in detail.

Hardware Sound Generation

The 8912 PSG is three channel device; that is, it can play three different sounds simultaneously; quite an improvement on 'BEEP'! As well as being able to produce simple tones, it can produce white noise and can also produce sounds whose loudness (or amplitude, to use the technical term) varies as the sound is being produced without the CPU having to do anything! It is a sophisticated device, but fairly easy to use. It is controlled by 16 registers, each of which controls some aspect of the behaviour of the PSG. Of these 16, 15 are directly concerned with sound production and the last one is used by the 128 to look after the RS232 and MIDI interface. The PSG can be used with the computer in either 48 or 128 mode; this is a useful fact because the PLAY commands aren't available from 48 BASIC and so if you're running in 48 mode you can still access the PSG from machine code. The business of producing sounds with a PSG is thus one of knowing how to program the PSG registers correctly, and that's what we'll look at next.

How do we get at these registers? Well, it would be very wasteful of address space to give each PSG register an individual address in the 128 I/O space. So, the PSG is accessed through just two I/O locations; one location, at I/O address 65533 is called the **ADDRESS WRITE** location and we use it to tell the PSG which register we want to write data to. The second location, at I/O address 49149 is called the **DATA WRITE** address and we use it to tell the PSG exactly what value we want to put in a register. It's also possible to read values back from the PSG by reading from I/O address 65533. These locations are accessed by use of the OUT (C),r instructions which we saw in Chapter 10. If you remember, in these instructions the full 16 bit I/O address, which is needed in this case, is stored in the BC register pair.

Writing to the PSG

This is done in the following way:

1. Write the number of the PSG register of interest to I/O address 65533 with an OUT (C),A instruction, or something similar, A holding the register number.

The function of each register will be explained soon. The register number will be between 0 and 15, the upper 4 bits of the A register being irrelevant.

2. Now send the data byte to be written to the register by writing it to I/O address 49149 in a similar fashion. The allowable values vary from register to register, as we'll soon see.

To make life easier, I use a small subroutine to write the values to the PSG. Call the routine with A holding the register number and H the data to be written to the PSG register. The subroutine is:

```
sound:  LD      BCm#FFFD
        OUT     (C),A
        LD      B,#BF
        OUT     (C),H
        RET
```

So, let's begin our examination of PSG registers by looking at those concerned with tone generation.

Registers 0 and 1

These are treated by the PSG as being a 12 bit register, the top 4 bits being in Register 1 and the lower 8 bits in Register 0. The upper 4 bits of Register 1 are thus not used. The value written to these registers controls the pitch of the tone played on Channel 1 of the PSG. The value held in the lower 4 bits of Register 1 has a larger influence on the pitch of the sound than the value held in Register 0, and for this reason Register 1 is called the Channel 1 **COARSE TUNE CONTROL REGISTER** and Register 0 is called the Channel 1 **FINE TUNE CONTROL REGISTER.** The lower the total value held in

these registers, the higher the pitch of the tone produced is. The PSG in the 128 is capable of producing tones with pitches between 27Hz and 100 000Hz, though anything above 20 000Hz is likely to be inaudible!!

To write values to these registers, we simply use a piece of code like:

```
LD      A,0
LD      H,12    ; value for R0
CALL    sound   ; routine shown above
LD      A,1
LD      H,12    ; value for R1
CALL    sound
RET
```

Registers 2 and 3

These are the pitch control registers for Channel 2 of the PSG Register 2 is the Fine Tune Control and Register 3 the Coarse Tune Control. They are used in a similar way to Registers 0 and 1.

Registers 4 and 5

These are the pitch control registers for Channel 3. Register 4 is the Fine Tune Register and Register 5 is the Coarse Tune Register.

Register 6

This isn't used in tone production, so it will be examined later.

Register 7

If the Accumulator is the most important register of the Z80 CPU, then this is certainly the 'accumulator' of the PSG, controlling whether sound is actually played on a given channel or not. Unless the bits in this register are set correctly, no sound will be produced by the chip. Each bit controls a

certain aspect of the PSG; bits 0 to 5 are concerned with sound, bit 6 with input/output and bit 7 isn't used by the PSG. Figure 13.2 shows how this register is arranged.

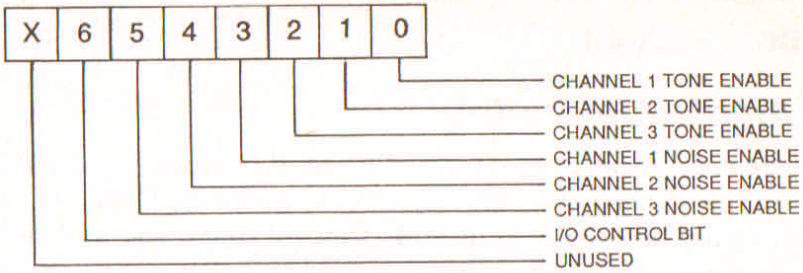


Figure 12.2

Bit 0 is the **TONE ENABLE BIT** for Channel 1. When set to 0, a tone can be played on Channel 1 of the PSG, and the Channel is said to be enabled. If set to 1 then no tone will be played on Channel 1 and it's said to be disabled.

BIT 1 is the Tone Enable Bit for Channel 2, and works in a similar way to Bit 0.

BIT 2 is the Tone Enable Bit for Channel 3, and works in a similar fashion to Bit 0.

BIT 3 is the **Noise Enable Bit** for Channel 1. As we've already said, the PSG can produce white noise as well as tone. This bit, when set to 0, will allow this noise to be heard on Channel 1. Setting both this bit and bit 0 to 0 will allow both tone and noise to be played on Channel 1. Setting both these bits to 1 will completely inhibit all sound output on Channel 1.

BIT 4 is the Noise Enable Bit for Channel 2.

BIT 5 is the Noise Enable Bit for Channel 3.

BIT 6 is the **I/O Control Bit**. One register of the PSG can be used to communicate with other electrical devices in the computer system rather than to produce sound. This I/O port can be used as an output, in which signals are sent from the PSG to other devices, or as an input, when the register receives

signals from other devices. This bit decides whether the port is to be an input or an output; when set to 0 the port is an output. In the Spectrum 128 system this port should always be an output, so leave this bit well alone!

BIT 7 isn't used.

Amplitude Control

The amplitude, or loudness, of a sound produced by the PSG is dependent upon the value in PSG Registers 8, 9 and 10. These are called the PSG Amplitude Control Registers, and there is one for each Channel. Register 8 controls Channel 1, Register 9 controls Channel 2 and Register 10 controls Channel 3. For the time being, we'll treat each register as a 4 bit register; this allows us 16 different levels of loudness for each channel. 0 is the lowest level (silence) and 15 is the highest level. The changes in volume from the PSG are not smooth. Instead, they are like steps, and you will be able to hear these steplike changes in amplitude as the amplitude of a sound changes. There is a fifth bit to each of these Registers, but we'll look at that later in the Chapter.

You'll be glad to know that we now know enough about the PSG to set up a Channel to play a tone! There are three stages to this process.

1. Set up the Pitch Control Registers to hold an appropriate value.
2. Set up the Amplitude Control Register to give the required volume.
3. Enable the tone for the Channel of interest.

Listing 13.3 shows a simple program that does this, and produces a tone on Channel 1.

```
ORG     60000
LD      A,0
LD      H,12
CALL    psg       ; load register 0
```

```
        LD      A,1
        LD      H,6
        CALL    psg     ; load register 1
        LD      A,8
        LD      H,15
        CALL    psg     ; register 8
        LD      A,7
        LD      H,62
        CALL    psg     ; register 7
        RET

psg:    LD      BC,#FFFD
        OUT     (C),A
        LD      B,#BF
        OUT     (C),H
        RET
```

Listing 12.3

The subroutine 'psg' is the same as the 'sound' routine discussed a little while ago. Line 120 of the program sets up register 7 so that bits 0, 6 and 7 are set to 0 and the other bits are set to 1. If you run this program, you will note one thing; the sound doesn't stop! When directly using the PSG we must always tell the chip when to stop. There are two ways of doing this:

1. Set the relevant bit of Register 7 to 1. This is the proper way.
2. Set the relevant Amplitude Control Register to 0

If you need to stop the PSG from BASIC, as you will want to if you've just run the above program, then the command PLAY "a","a","a" will do the trick. This works by using the BASIC interpreter to send a tone to each channel for a given length of time. This over-rides whatever the PSG was previously doing and so stops the sound.

Listing 12.4 shows how we can use the PSG to produce a tone for a given length of time. Assemble to 60000 and try the routine out. You could also try altering the values put into Registers 0 and 1 to alter the pitch.

```
        ORG     60000
        LD      A,0
        LD      H,12
        CALL    psg
        LD      A,1
        LD      H,6
        CALL    psg
        LD      A,8
        LD      H,15
        CALL    psg
        LD      A,7
        LD      H,62
        CALL    psg
        LD      B,100     ; delay of 2 seconds
delay:  HALT
        DJNZ    delay
        LD      A,7
        LD      H,63
psg:    LD      BC,#FFFD
        OUT     (C),A
        LD      B,#BF
        OUT     (C),H
        RET
```

Listing 12.4

The main code of interest in this routine is that used to produce a time delay, between lines 140 and 160. The HALT instruction is executed 100 times. HALT causes the CPU to stop what it's doing until an interrupt comes along; unless the interrupts have been disabled, this will occur every 1/50th of a second. So, we get a total delay of 100/50 seconds.

Once a tone is being played, we could alter the Amplitude Control Register for that channel to produce a 'fade out' effect

rather than an abrupt finish to the tone. This could be done by simply decreasing the Amplitude Control Register by 1 every half second or so; the delay used could be similar to that used in listing 12.4. However, I'll leave that one for you to sort out! It's also possible to alter the value in the pitch control register whilst a tone is being played, thus getting a constantly changing tone. Listing 12.5 shows how this can be done.

```
        LD      A,0
        LD      H,16
        CALL    psg
        INC     A
        CALL    psg
        LD      A,8
        LD      H,15
        CALL    psg
        LD      A,7
        LD      H,62
        CALL    psg
        LD      H,16
ol:     LD      B,100
del:    HALT
        DJNZ    del
        LD      A,1
        CALL    psg
        DEC     H
        JR      NZ,o1
        LD      A,7
        LD      H,63
psg:    LD      BC,#FFFD
        OUT     (C),A
        LD      B,#BF
        OUT     (C),H
        RET
```

Listing 12.5

You might like to try altering these programs to play tones on the other channels as well as Channel 1. All you have to do is

set up the pitch tune registers and the amplitude control register for the Channels that you want to use. Then simply set the appropriate bits of Register 7 to enable the channels.

So far, we've looked at the tone generating abilities of the PSG; let's now look at the noise producing abilities of the PSG. Some unpleasant types might say that ALL the sound produced by the computer is a 'noise'! When we talk about noise, though, we're actually talking about a particular type of sound which contains sounds of many different pitches. This sound is often called **WHITE NOISE**, and sounds like a hissing or roaring noise. It's called White Noise because it's similar to normal 'white light', which is made up of light of all different colours ('pitches'). Many natural and man made noises have a high proportion of White Noise in their make up. Examples are rainfall, gunfire and explosions. Noise can also be used to produce simple percussion effects, such as snare drums.

Noise can be layed on any of the three channels of the PSG, or all of them if you want. The amplitude of noise on a channel is set by the value in the Amplitude Control Register for that particular Channel. Thus it's not possible to play tone and noise on the same channel at different levels of volume. As with tone, before you can hear a noise on a channel the relevant bit of Register 7 must be set to 0. So, to enable noise on Channel 1 we set bit 3 of Register 7 to 0. To see this in action, try the below listing.

```
        LD      A,8
        LD      H,15
        CALL    psg
        LD      A,7
        LD      H,#37
        CALL    psg
        RET
psg:    LD      BC,#FFFD
        OUT     (C),A
        LD      B,#BF
        OUT     (C),H
        RET
```

The above routine simply enables the noise on Channel 1. Just as a tone has a pitch, so does noise. The pitch of white noise is measured in terms of the relevant amounts of low and high pitched sound in the white noise. Highly pitched noise is characterised by a 'hissing' noise, and low pitched noise is more of a roaring, rushing noise. Register 6 is the PSG register responsible for noise pitch; as there is only one register and three channels, it's clear that each channel will play noise at the same pitch, dependent upon the value in Register 6. Register 6 is a 5 bit register, thus giving legal values in the register of between 0 and 31. A value of 0 gives the highest pitched noise and a value of 31 gives the lowest pitched noise.

We'll now go on to look at what is probably the most sophisticated aspect of the 8912 PSG; its ability to apply ENVELOPES to the sound produced.

Envelopes

First of all, in sound production envelopes are nothing at all to do with sending letters through the post!

What's the difference in sound, for example, between a Middle 'C' played on a piano and the same note being played on a flute? The pitch is the same, but the way in which the amplitude of the sound varies with time is totally different. Although this isn't the only reason that the notes sound different, it's quite important. The piano note starts off at quite a loud volume, then decays away as the note is played. The flute note stays at roughly the same volume as long as someone is blowing it. The way in which the amplitude of a sound varies with time is called the **AMPLITUDE ENVELOPE** of a sound. Figure 12.3 shows a couple of envelopes; 13.3a shows the typical tone played by the PSG, with a very abrupt beginning and end to the sound. 12.3b shows an envelope which will cause the sound to gradually increase then decrease.

You can probably see that we could produce an amplitude envelope like this by simply altering the relevant Amplitude

Control Register. However, this would require the full attention of the CPU, unless some complicated interrupt routines were written to cause the CPU to alter the register every so often. The PSG, though, will look after the production of an envelope automatically. We can program the PSG to apply one of 8 different Amplitude Envelopes to the sound, whether tone or noise, being produced on one of the Channels. The envelopes thus applied are called **HARDWARE ENVELOPES** because they are produced by the PSG alone with no further instructions from the PSG after the CPU has started the sound off.

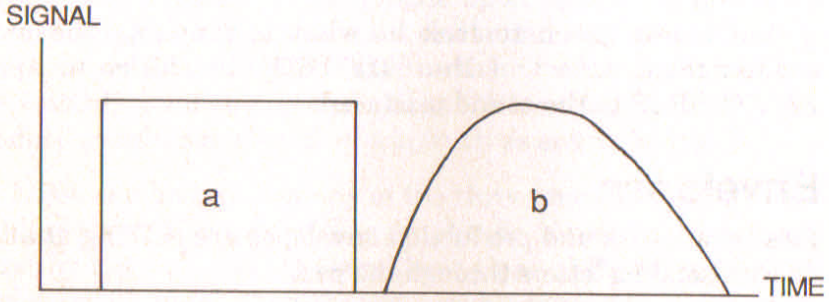


Figure 12.3

Register 13

This register tells the PSG the type of Envelope that we want to apply to a channel. It's called the **Envelope Shape Control Register**, and as there is only one of these it's clear that each channel must play sound using the same envelope. The register is a 4 bit register. However, this doesn't give us 16 different envelopes. In fact, there are, as just mentioned, 8. The others are all duplicates. Figure 12.4 shows the different hardware envelopes that are available, and also the values that need to be written to this register to get each envelope. Putting any other values in Register 13 will cause one of these Envelopes to be used.

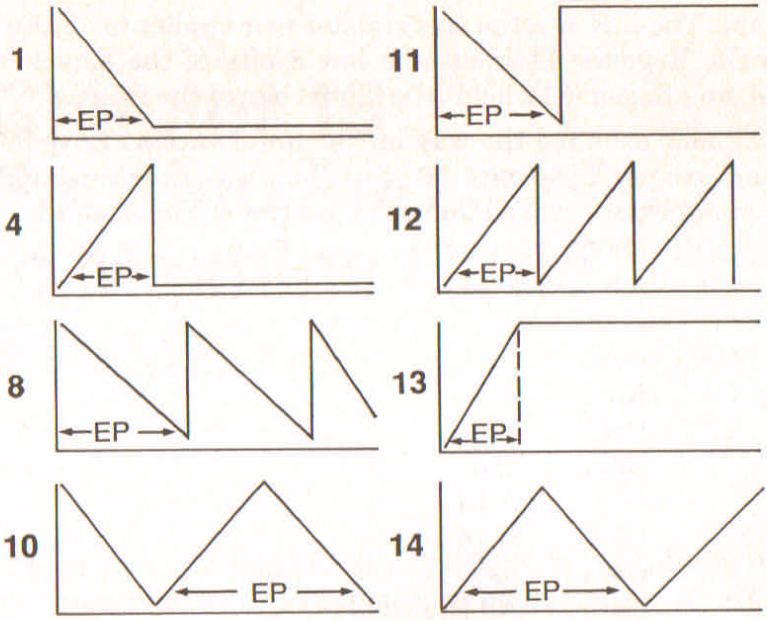


Figure 12.4

So, how do we tell the PSG that we want to apply an Envelope to a sound being played on a given Channel rather than allowing the sound to be played with the Amplitude set by the relevant Amplitude Control Register?

This is where bit 5 of each Amplitude Control Register comes in useful; if this bit is set to 1, then the sound on that channel will be played under the direction of the Envelope whose number is in Register 13. If set to 0, then the sound on that Channel will be played at the amplitude specified by the lower 4 bits of the Amplitude Control Register. So, setting Register 8 to 16 will cause the sound played on Channel 1 to be played under control of the Envelope whose number is in Register 13.

Registers 11 and 12

These registers are responsible for setting the **ENVELOPE PERIOD** (EP in Figure 12.4). This is a measure of the length of time taken for the PSG to execute an envelope. The larger that this 16 bit value is, the longer it will take to execute the

envelope. The period set in this register pair applies to all three channels. Register 11 holds the low 8 bits of the Envelope Period, and Register 12 holds the high 8 bits of the value.

Let's now examine the way in which the various Envelope registers are used. Listing 12.6 shows how we can set up a tone on Channel 1 to be played under the control of Envelope 14.

```
ORG  60000
LD   A,8
LD   H,16
CALL psg
LD   A,1
LD   H,4
     CALL   psg
     LD     A,13
     LD     H,14
     CALL   psg
     LD     A,11
     LD     H,128
     CALL   psg
     INC    A
     CALL   psg
     LD     A,7
     LD     H,62
psg: LD     BC,#FFFD
     OUT    (C),A
     LD     B,#BF
     OUT    (C),H
     RET
```

It's possible to get a wide range of sound effects by playing around with Envelope Periods. The production of sound effects, whether from BASIC or machine code, is always a 'trial and error' affair.

Machine Code PLAY Commands

The Spectrum 128 gives us a very easy way of using the PSG with the PLAY command. It is also possible to access the ROM routines that do this from machine code, but this is a little beyond the scope of this book.

Chapter 13
Passing Parameters to Machine Code Programs

Although we can use POKE and PEEK to get information to and from addresses in memory that our machine code has used, it would be rather nice if we could get access to variables that BASIC uses and thus pass information between BASIC and machine code in these variables. Well, it can be done, but there are complications. In this Chapter we'll look at how we can find out where in memory particular variables are, and how the variables themselves are arranged. However, the difficulties start when we find that all the numeric variables in the Spectrum are stored as 5 byte **FLOATING POINT** numbers; a special coding system that allows us to represent decimal and fractional numbers. I won't be covering this coding system in this book, but later in the book we'll look at ROM routines that allow us to convert these floating point numbers, in some cases, into 8 or 16 bit numbers.

VARS

The variables used in a BASIC program are stored just after the BASIC program. A few moments thought will show that this address will move as programs of different length are loaded into the computer. For this reason, the address of the start of the area of RAM used by for variable storage is held in

a two byte System Variable called VARS. This is at address 23627 and 23628, the low byte in 23627 and the high byte of the address in 23628.

Structure of the Variables

Within the area of memory that is assigned for variable storage, the variables are stored in the order in which they were created by the program. So, if you create the variable 'a' as the first variable in your program the address held in VARS will point to the first byte used to store the information about variable 'a'. As more variables are added, the information about them is simply 'tacked on' to the currently existing variable information until the memory gets full or until you CLEAR all the variables. Of course, different types of variable are stored in different ways in RAM; the main types of variable are as follows:

1. Numeric variables whose name is one letter long.
2. Numeric variable whose name is more than one letter long.
3. Array of numbers.
4. FOR...NEXT control variable. e.g., in the statement FOR i=0 TO 100 'i' is the control variable.
5. String of characters.
6. String array.

Of these different variable types, type '4' is of minimum use, as there is little point in altering such a variable from within machine code. Of the other types, I shall concentrate on using numeric variables with single character names and strings. The others are useful, but I suggest you get some experience with the simplest variable types first. Figure 13.1 shows how these two types of variable are arranged in memory; the numeric variable and the string variable with a single letter name.

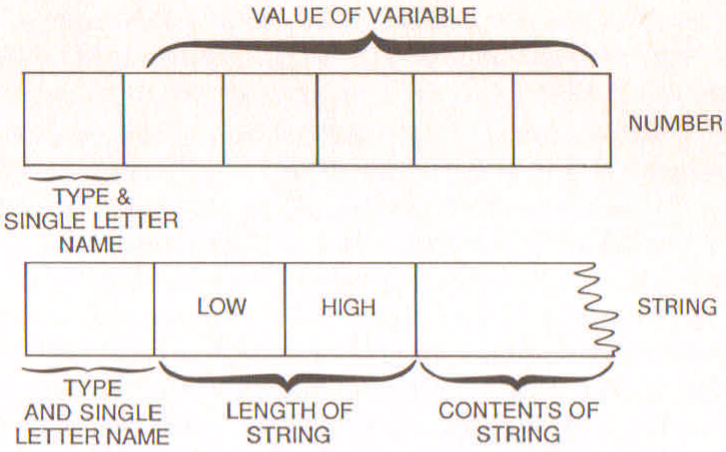


Figure 13.1 Variable Structures.

As can be seen, the first byte of these variable structures contains two vital pieces of information; the name of the variable (a single, coded, lower case letter) and the type of the variable. The variable type is held in the top three bits of this byte. Obviously, the other variable types have different 'type bits' but they still use the three most significant bits of the first byte of the variable.

As a simple demonstration of the storage of variables, try the below program. (Listing 13.1)

```
10  LET a=0
20  LET a$="Hello There"
30  LET vars=PEEK(23627)+ 256*PEEK(23628)
40  FOR i=vars TO vars+30
50  PRINT i;"  ";PEEK(i);"  ";CHR$ (PEEK(i))
60  NEXT i
```

Listing 13.1

The first address accessed will hold the value 97; this is the coded variable name of 'a' in line 10. To extract the variable name from this type of variable, the following steps are needed:

1. AND the byte with 00011111. This will get rid of the type bits (which are set to 011 in this case).

2. Add 96 to the result. This will give the ASCII code of the lower case version of the single character variable name.

The next 5 bytes are the Floating Point representation of the value assigned to a in line 10. Try different values in line 10 if you want to, and see how the values in the Floating Point representation. Don't worry too much about the values of these bytes here. I'll explain ways of getting the values held in these bytes into registers in Chapter 16. That completes the representation in memory for the numeric variable 'a'. Now comes the representation for the string variable, a$. The first byte of this is 65; this is a coded form of the variable name, and the steps for extraction of the variable name are as follows:

1. AND with 00011111 to get rid of the type bits, which are 010 in this instance.
2. Add 96 to the result; this will again give the ASCII code of the lower case version of the single letter variable name.

Then follow two bytes that hold the length of the string, stored in traditional Z80 fashion with the low byte first. Then comes the bytes that represent the ASCII code of the string. These are not coded in any way.

Find the Variable

This is a bit like the card game of 'Find the Lady'; you know she's in there somewhere, but it's not so easy finding her! In the example above, the variables were easy to find because they were the first two variables in the variables area of memory. However, in a real program we might wish to pass several different variables over to a machine code program that have been declared at different points in the program; it's clear that we need a method of searching through the variables area of memory until we find the variable of the name AND TYPE that we're looking for. So, we must look at the first byte of each variable name in memory and check its type. If the type is OK, then we look at the name of the variable. In the case of the two

simple variables listed above, it's quite easy as we're only looking at variables with a single letter in their name. If the name isn't correct, then we go on to look at the first byte of the next variable. It's this last bit that causes the problems; different variable types occupy different numbers of bytes in memory. To get to the first byte of the next variable we must add the length of the variable that is currently being examined to the address of the first byte of the current variable. This would take a little working out; however, we can take advantage of a ROM routine called the 'NEXT ONE' routine that will allow us to find the address of the first byte of the next variable, assuming that we're already on the first byte of a variable. This isn't too difficult, as we already know that the system variable VARS holds the address of the first byte of the first variable.

The routine is at address &19B8, and is used in the following way:

1. Before calling the routine, HL should point to the first byte of a variable structure in memory.
2. The ROM routine is then called.
3. On exit from the ROM routine, the address of the NEXT variable in memory is stored in the DE register.

As an example of the use of this routine to find the first byte of a particular variable, look at listing 13.2. Here we're trying to find the variable 'a$' in memory. Assemble the program and then issue a 'LET a$="Hello There"' instruction to assign a value to a$. PRINT USR 60000 will then return the address of the first byte of memory that represents a$. Note that this program doesn't generate any error if the variable hasn't been set up. You might have problems if this happens, so take care. The 'CP 65' instruction in the program checks the first byte of each variable found by the ROM routine to see if it's the first byte of 'a$', which will have the value 65. How do we arrive at this value?

Well, looking at Figure 13.1 you'll see that for a string variable the top 3 bits are 010. The lower 5 bits of the first byte

represent the ASCII code of the lower case version of the single letter variable name, minus 96. Putting this together, for a$, the lower 5 bits will hold the value 97-96, or 1. To this we add the value held in the upper three bits, in this case 64, thus getting the final result of 65. So, on with the program...

```
        ORG     60000
        LD      HL,(23627)
loop:   LD      A,(HL)
        CP      65
        JR      Z,found     ; go if found
        CALL    &19B8       ; find next variable
        PUSH    DE          ; put address in to
        POP     HL          ; the HL pair
        JR      loop        ; round again
found:  PUSH    HL
        POP     BC          ; address in BC for
        RET                 ; return to BASIC.
```

Listing 13.2 Finding Variables.

Once we've got the address of a variable, we can obviously alter its contents or get the values from the variable to use in our own programs. For example, the below listing shows a simple program to return the length of 'a$' to BASIC.

```
        ORG     60000
        LD      HL,(23627)
loop:   LD      A,(HL)
        CP      65
        JR      Z,found
        CALL    #19B8
        PUSH    DE
        POP     HL
        JR      loop
found:  INC     HL
        LD      B,(HL)
        INC     HL
        LD      C,(HL)
        RET
```

Listing 13.3 String Length.

For numeric variables, the only simple way to get access to the values in them is to put the values on to the Floating Point Calculator stack and then recover the value from the stack into a register pair. There are further details of the Floating Point Calculator in Chapter 16. However, for now, we'll use a ROM routine to do this job. The routine is at address &33B4, and is called with BC holding the value 5 and HL holding the address of the first byte of the representation of the number. So, if we've used the above routine to get the address of a numeric variable with a single letter variable name, adding one to the resultant address will produce the address to be passed to the routine at &33B4.

Chapter 14
Keyboard Operations

In this Chapter we'll examine the way in which we can read the Spectrum keyboard from a machine code program. I said sometime ago that the keyboard is read by the Interrupt routine called 50 times a second by the CPU. All we need to be able to do is get access to the information that is read in to the computer when the keyboard is read. Of course, we could actually read the keyboard directly, looking at the I/O address that controls it, but in this book I'll simply explain the use of the keyboard in such a way that a character code is returned when we read the keyboard. This is quite simple, because when a key is pressed the Operating System updates a couple of System Variables to hold the key pressed and to indicate that a key has been pressed.

The system variable **LASTK** at address 23560 holds the key code of the last key pressed; however, this might have been pressed 10 minutes ago! It is only updated when a new key is pressed. We can thus have difficulties in knowing if a value in LASTK is 'new' or 'old'. Fortunately, a further system variable at address 23556, part of an 8 byte system variable called **KSTATE**, can be used to see if a key is being pressed at a particular instant in time. If a key is NOT being pressed, then address 23556 has the value 255. If a key is being pressed, then this variable will have a value that depends to some extent on

which key is being pressed. We can thus simply check this byte for a value of 255 whenever we read a value from LASTK; if the value in 23556 is equal to 255 then the value in LASTK is an 'old' key which can be ignored. Otherwise, it's a 'new' key and so you can act on it.

Alternatively, you could write a short subroutine to wait for a key press and return the keypress to you in the A register. The simplest routine that I can come up with is:

```
keypress:  LD    A,(23556)
           INC   A
           JR    Z,keypress
           LD    A,(23560)
           RET
```

The only time that the INC A would set the Z flag and cause the relative jump back to the beginning of the routine is when the A register holds 255; that is, when a key isn't being pressed.

Once a key is returned, you can use CP instructions to see what the key is and act on it. One problem that you might have is if the keyboard is in Extended or graphics mode when you do this; you might get an unexpected value returned. A more common problem is that of the CAPS-LOCK being off when you expect it to be on, or vice versa. Setting bit 3 of location 23568 to 1 will set the CAPS-LOCK on, and setting it to 0 will set the CAPS-LOCK off.

The only other things we can do with the keyboard are to affect the rate at which keys 'repeat', and the noise that is made when keys are pressed. The keyboard 'click' is set by the value in location 23609. Increasing the value will give a longer 'click' and shortening it will decrease the click length.

Keyboard Repeats

The keyboard is arranged so that if you hold a key down for a brief time it will start repeating, producing more 'characters' without you having to take your finger off the keyboard. In the UK, the time before the key starts repeating, the REPEAT

DELAY, is 35 50ths of a second. In countries with 60Hz mains electricity it's 35 60ths of a second. Once repeating, the key will repeat once every tenth of a second or thereabouts (the REPEAT RATE) until you take your finger off the key.

The Repeat Delay is stored in address 23561, and the Repeat Rate is stored in address 23562.

It's occasionally useful, in games programs, for example, to decrease the repeat delay, by reducing the value in address 23561, and increase the repeat rate by increasing the value in address 23562.

Chapter 15
The Screen Display

The Spectrum supports a screen layout of 24 rows of 32 characters; a rather odd screen layout compared to some machines, but quite useable. The only problem with this screen layout is that it's a little awkward to access directly from machine code instructions. You can load bytes into screen memory using any of the instructions that we've seen already, but the screen RAM is arranged in a way that isn't immediately obvious. For this reason, I will concentrate in this Chapter on using the ROM routines that are available to write information on to the screen, but I will give some brief details of the layout of the screen RAM. So, here goes...

Screen RAM Layout

The RAM used to hold the information that constitutes that screen image is a block of 6144 bytes of memory starting at address 16384. It is arranged as 192 'rows', each row containing 32 bytes. A 'character line' on the Spectrum screen is defined by 8 of these screen memory rows. Each 'bit' in these bytes represents a dot, or **PIXEL**, on the screen; this gives the Spectrum a pixel resolution of 256 (32*8) by 192 pixels. For some reason, probably quite logical (though I can't actually work out what it is!) the rows of pixels are not arranged

consecutively in RAM. For example, we might expect pixel row 2 to be stored immediately after pixel row 1, the first 256 bytes of video RAM thus defining the first character line on the screen. This isn't so. In fact, the top row of character line 0 is stored first, then the top row of character line 1, then the top row of character line 2, and so on down to character line 7. Then the second row of line 0, the second row of line 1, and so on, finally ending up with the bottom pixel rows of character lines 0 to 7. This is then repeated for screen character lines 8 to 15 and finally for screen character lines 16 to 23. The screen RAM can thus be seen to be made up of three 2k blocks of memory, each block defining a portion of the screen. (Figure 15.1) The top left of the screen is address 16384. Clearly, getting the address of the byte of screen RAM that holds the pixel corresponding to a given x,y coordinate on the screen isn't a trivial task, and it's a little beyond the scope of this book. If you're interested, a method is described in *Advanced Spectrum Machine Language* by David Webb, also published by Melbourne House.

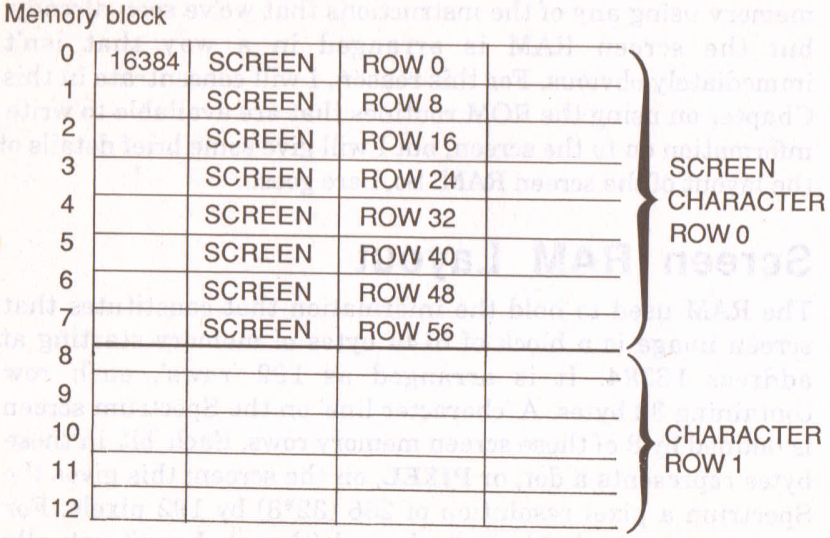


Figure 15.1. Screen RAM Layout

The block of memory described above simply defines whether or not a pixel is to be 'set' to the ink colour (bit set to 1) or 'cleared' to paper colour (bit set to 0). The colour of the pixel isn't set by this block of memory; if you think about it, this is quite logical. A bit can have either a value of 1 or 0; we'd need 3 bits to define the seven colours available on the Spectrum, and so if this same block of memory were to be used to define the colour of plotted pixels as well as their position, we'd have a lower resolution (fewer individually addressable pixels) screen. The colour of the pixels is specified by a smaller area of memory called the **ATTRIBUTES FILE.**

The Attributes File

This is a block of memory found between 22528 and 23295 inclusive. Each byte in this block defines the colours used as INK and PAPER, and the BRIGHT and FLASH characteristics of a single CHARACTER square, NOT a pixel. Thus, while we can have single pixels set on or off, we can only alter the attributes (colour, flash on or off, etc.) of character squares. This 'lower resolution' for attributes can be an inconvenience. The character squares of the screen map on to the bytes in the attribute file in a logical fashion. The first 32 bytes of the attribute file hold the attributes for the first character row, the next 32 bytes the second character row, and so on. The attribute bytes are split into 4 separate parts; 3 bits define the INK colour, 3 bits the PAPER colour, 1 bit the FLASH parameter and 1 bit the BRIGHT parameter. (Figure 15.2)

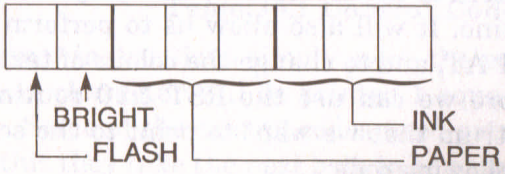


Figure 15.2

The various attributes are coded as follows:

BRIGHT	1=ON 0=OFF	FLASH	1=ON 0=OFF

COLOURS	
000	Black
001	Blue
010	Red
011	Magenta
100	Green
101	Cyan
110	Yellow
111	White

Thus to set the attributes of a character square on the screen to flashing red on white we'd set PAPER to 7, INK to 2, FLASH to 1 and BRIGHT to 0. This would give an attribute byte of:

```
10111010
```

Setting the attributes directly by loading values into the Attribute File doesn't affect any image that's been drawn to the screen, and so provides a means of altering the attributes of things that we've already drawn on the screen without having to redraw them.

OK, so we've seen how the screen of the Spectrum is arranged. How can we get something on to the screen? Well, we're fortunate in that we've got a couple of ROM routines that will allow you to put text and graphics on the screen. I'll only be scraping the surface here; again I direct you to *The Complete Spectrum ROM Disassembly* for further details of these routines.

Printing Characters

We can print characters to the screen very easily by using the RST &10 ROM routine. It will also allow us to perform such operations as PRINT AT, and to change the colour of text that we're printing. Before we can use the RST &10 routine we should tell the Spectrum that we want to print to the screen. This is done by the following code:

```
LD      A,2
CALL    &1601
```

This need only be done at the start of any printing, not before each and every character is printed! The parameter in the A register is called the 'channel' number; 2 simply indicates the screen.

To actually print a character we simply use:

```
LD      A,code
RST     &10
```

where 'code' is the character code of the character that you want to print on the screen. Codes with a value less than 32 will have either no effect at all or will allow you to, for example, change the INK colour for this printing operation, do a PRINT AT, and so on. These are the CONTROL CODES, called this because they control a particular facet of the computers display. Table 15.1 shows the effects of some of the control codes.

Code	Extra bytes needed	Effect
8	0	Cursor Left
9	0	Cursor Right
10	0	Cursor Down
11	0	Cursor Up
13	0	Pressing ENTER
16	1	INK n command
17	1	PAPER n command
18	1	FLASH n command
19	1	BRIGHT n command
21	1	OVER n command
22	2	AT y,x command

Table 15.1 Control Codes

The cursor instructions simply move the position at which the next text character will be printed 1 square in the directions shown. Some of the control codes need extra information, and to get this they take the next byte to be passed through the RST &10 routine. Thus after a call to RST &10 with A=17, then next byte to be put in A to be used by RST &10 will not be treated as a character code; instead, it will be treated as the parameter needed by the PAPER instruction given by

'printing' the control code 17. AT needs two extra parameters, the first for the Y coordinate and the second for the X coordinate. The PAPER, INK etc. attributes set up by using RST &10 in this way are called TEMPORARY attributes. This is because they only operate within the printing that is currently being done, unlike the PERMANENT attributes set by the INK and PAPER commands proper. To see the RST &10 routine in action, try the below piece of code out. It prints a Blue letter A on yellow paper.

```
LD      A,2
CALL    &1601
LD      A,16
RST     &10
LD      A,1
RST     &10
LD      A,17
RST     &10
LD      A,6
RST     &10
LD      A,65
RST     &10
RET
```

If you've got a lot of printing of characters to do, it's often useful to use a 'string printing subroutine' to replace all the LD A and RST &10 instructions. There is one of these in the ROM of the computer, at address &203C. DE is set up to point to the first character to be printed, and BC holds the number of characters to be printed. For example:

```
        LD      DE,message
        LD      BC,11
        CALL    &203C
        RET
message: DEFM   "Hello There"
```

Alternatively, you could write your own. The simple subroutine listed below is used in the following manner. HL should point to the first character of the string, and the string should finish with a byte set to 0.

```
print:  LD      A,(HL)
        CP      0
        RET     Z
        PUSH    HL
        RST     &10
        POP     HL
        INC     HL
        JR      print
```

There are other useful ROM routines that we can use. To print numbers out, we can use a routine at address &2DE3. This prints out the last value on the Floating Point Calculator Stack. For further details of this, see Chapter 16. A call to 3582 (decimal) will scroll the display by 1 line. Finally, the routine:

```
cls:    LD      A,2
        CALL    &1601
        CALL    &06DB
        LD      A,2
        CALL    &1601
        RET
```

will clear the screen.

Permanent Attributes

We can simulate the permanent INK, PAPER, BRIGHT and FLASH instructions by altering the value of a system variable at address 23693. This holds a copy of the attributes that are to be applied to the whole screen, and attributes set up like this will be used by future print and CLS operations. The arrangement of bits in this byte is the same as for the bytes of the attribute file, and you can set up colours of INK and PAPER in the same way.

Extra Character Sets

The normal character set used by the computer, that is, the group of characters that can be printed to the screen, is held in the ROM in the computer. Some of the characters, the User Defined Graphics, are held in RAM, but the rest are in ROM.

What this means is that when the computer wishes to print a character on the screen the actual bytes to be loaded into the screen RAM to produce a given character are read from the character definitions in ROM or RAM. The character definitions are 8 bytes long, each byte defining the pixels in one screen row of the character. Now, it's possible to redefine some of the characters of the character set under normal conditions; after all, that's what User Defined Graphics are for.

It is also possible to redefine the WHOLE of the character set by telling the computer to get the character definitions from a different part of memory to that from which the definitions are normally read. The address of the first byte of the definition of the 'space' character is held, slightly modified, in a system variable at addresses 23606 and 23607. This variable holds the address of the first byte of the definition-256. By altering the value in this system variable we can change the area of memory used for character definition.

Simple Graphics

I'll only give one simple graphics routine; this is the ROM routine used to PLOT a single point on the screen. The routine is called at address &22E5 with the X coordinate in the C register and the Y coordinate in the B register. The last X coordinate used by the plotting routines is stored in address 23677, and the last Y coordinate used is stored in 23678.

Chapter 16
ROM Routines

The Spectrum 128/Plus 2 has a ROM which hasn't altered to any great extent from the early days of the 16k Spectrums which were the first sub £100.00 colour computers. This is quite useful to us, because in the intervening period of time the ROM has been DISASSEMBLED- that is, the various routines that make up this large program have been analysed and documented. *The Complete Spectrum ROM Disassembly* by Ian Logan is published by Melbourne House and this book documents the routines with the BASIC ROM of the 128/Plus 2 but NOT the routines in the Editor ROM.

Why is this useful? Well, it allows us to call subroutines that are in the ROM of the computer from our machine code programs. The advantages of this approach are as follows:

1. Our machine code programs are shorter, as the ROM code is in the machine anyway.
2. It's easier for us to write our programs. We don't have to write such fundamental routines as a 'print a character to the screen' routine.
3. Professionally written ROM code might be a little more reliable than our own, certainly at the beginning of a machine code programming career.

There are a couple of drawbacks to using ROM code, and these are:

1. You may not know the address in the ROM to 'call' to get the desired results. Even if we have access to a ROM listing, some of the more complex routines can be called at a variety of addresses, each of these ENTRY POINTS allowing the routine to do a slightly different job.

2. If the manufacturer of the machine alters the ROM after you've bought your computer, then your machine code programs may not run on your friends' computer if you've called ROM routines. In the later version of the ROM the routines may be at different addresses. This is particularly important if you're writing programs for sale!

3. ROM routines occasionally have undocumented 'bugs' which might cause you some very 'odd' problems.

4. ROM routines may be written to do a variety of jobs; a typical example of this is the RST &10 instruction which is used on the Spectrum to print out characters on the screen. Because it has to check for control characters, User Defined Characters, etc. as well as normal keywords, it has a lot of code in it that isn't really needed if all we want to do is to print out printable characters. Thus, this slows things down a little, and ROM code might not be as efficient as if you'd written the code needed with no extra frills.

The intent of these warnings is NOT to put you off using the ROM routines available, but to point out a few pitfalls. The following list of ROM calls is nowhere near comprehensive, but are some of the more useful ones. If you use them, the addresses refer to the BASIC ROM; this ROM will be paged in whenever a BASIC program is being executed, or when a BASIC instruction is being executed. If you have paged out the BASIC ROM for any reason, make sure that it's paged in before calling any of these routines.

The Error Restart

This routine is used by BASIC to generate errors; we can use it in a similar fashion to generate errors from our machine code programs. It is used in the following fashion:

```
error:  RST   &8
        DEFB  n
        RET
```

where 'n' is a single byte which is used to specify the error message to be generated.

The 'Print a Character' Restart

This routine, called by RST &10, allows us to print a character or control code to the screen. It is the routine used by the BASIC ROM to print out text . More details of this call were given in Chapter 15, along with other screen handling routines.

The Tape Routines

You can use the ROM routines that are responsible for saving and loading information on tape from your own machine code programs as well. The method shown here is very crude, in that:

1. File names are not used.
2. The exact number of bytes to be saved or loaded must be known.

Saving bytes is quite easy. The ROM routine at address 1218 (decimal) allows us to save a block of bytes to tape as data. To do this, it needs two things, the address of the first byte of memory to be saved and the number of bytes to be saved. Also, we have to tell the ROM routine that we're saving these bytes as 'data' rather than as a HEADER. The Spectrum usually saves its tape files in two parts; a header containing things like file name, length of file and type of information in the file, and a data block, which actually contains the data from memory. Using the ROM routines directly, it is possible to have a header

as well as a data block, but here we're simply saving data. We indicate this fact to the ROM routine by loading A with 255 before calling the routine. The address of the first byte to be saved is passed to the routine in the IX register, and the number of bytes is passed in the DE register pair. So, to save the 1000 bytes of memory starting at address 16384, we'd use a program like:

```
save:   LD      IX,16384
        LD      DE,1000
        LD      A,255
        CALL    1218
        RET
```

This routine will save the bytes on tape but you won't be able to load them back in without using the 'load bytes' ROM routine directly. To load bytes in, the registers are set up before calling the routine in the same way as above; here, though, IX holds the address to which the first byte is to be loaded into memory from tape and DE holds the number of bytes to be loaded. The load routine is called at address 1366.

Beep Routine

This ROM routine, to generate an audio tone, was covered in Chapter 13.

The Floating Point Calculator

This is a very large and complicated machine code routine, and the best that I can do here is to just 'scrape the surface' of its use. This routine is responsible for all the calculations performed by the BASIC ROM on our variables. Earlier on in the book, I mentioned that floating point numbers were a little complicated to handle; this is quite true, but by using some of the Floating Point Calculator routines we can at least do something with them, or even perform complicated arithmetic operations on 8 or 16 bit numbers held in Z80 registers.

The Floating Point Calculator (henceforth known as FPC) is used by issuing a RST &28 instruction followed by a sequence of data bytes. For example:

```
calc:   RST    &28
        DEFB   n1
        DEFB   n2
               ...
        DEFB   &38
        RET
```

n1 and n2 are parameters for the RST &28 instruction, and tell the FPC what it is to do next. Once you've given the FPC a complete sequence of arithmetic operations to do, you finish the list of FPC 'instructions' off with a DEFB &38, which simply tells the FPC, 'All done!'. It's a little like having a pocket calculator; we 'press buttons' on this 'calculator' by issuing DEFB instructions, and, rather than pressing the '=' key we issue a DEFB &38 instruction to finish everything off.

The FPC relies for its operation on a stack, similar to that used by the Z80 CPU. However, it's not the same stack; whereas the CPU stack operates on 16 bit numbers, the FPC stack works in 'words' of 5 bytes long. This is so that you can store a Real Number, which is 5 bytes long, on the stack in one 'go'. The FPC stack is stored in memory between the end of your BASIC variables and the Z80 CPU stack, and its exact position in memory is stored in a system variable at addresses 23651 and 23652, in the usual 'low byte first' arrangement. The stack is used in the same way as the CPU stack; that is, a value is pushed on to it and can be 'popped' off from the stack. The last item to be pushed on to the stack is always the first item to be popped from it. The 5 byte value held at the 'top' of the stack, that is, the next item to be popped off the stack, is called the LAST VALUE. Clearly, when the BASIC ROM wants to push or pop information from this stack it can't just use a single register pair; in fact, 2 register pairs and the A register are needed.

I'll now have a look at some of the simpler ways of using the Floating Point Calculator. If you want more details, have a look at *The Complete Spectrum ROM Disassembly*.

Putting a number on the stack

The easiest way is to load the number into the BC register and then call a ROM routine at address 11563. This will then put the value in the BC register on to the Floating Point Stack as a sequence of 5 bytes. The value in BC represents a number between 0 and 65535, rather than a Twos Complement number. The number thus put on the stack will be the 'last value' on the stack.

Recovering the 'last value'

Once we've got a last value on the stack, we can, if it's in the range of values that can be represented in a two byte number, get the last value off the stack in to the BC register. The ROM routine to do this is at address 11682. The routine is called and on return BC holds a value. If C=1 on return from the ROM routine, then it indicates that the number on the stack was too large. If Z is reset then the value returned in BC is a negative value. Obviously, if the value on the stack had a decimal part to it, the BC register would contain an integer version of the number. The below routine simply puts the BC register on to the stack, alters BC, then recovers the value from the stack before returning to BASIC.

```
ORG     60000
LD      BC,1000
CALL    11563
LD      BC,0
CALL    11682
RET
```

Use the routine from BASIC with PRINT USR 60000. '1000' should be printed to the screen.

This is all very well, but what about doing some sums with the values we can put on the stack? Let's see how we can do that next.

Calculations

The FPC performs calculations with numbers that we've put on the stack. For example, when it needs to add two numbers

together, the two numbers to be added are placed on the stack, and the addition 'instruction' causes them to be added and the result left on the stack as the 'last value'. This may appear to be a rather odd way of doing things, but is reasonably easy to use once you've had a little practice. As I've already indicated, the instructions to the Floating Point calculator are passed to it as 'DEFB' statements immediately after the RST &28 instruction. There are quite a few of these, and far too many to cover in this book. I'll thus confine myself to the four arithmetical operations of addition, subtraction, division and multiplication. Obviously, some sums will be more easily performed using the 8 or 16 bit arithmetic instructions of the CPU, but the techniques needed to use these simple operations on the Floating Point Calculator will be useful if you perform more advanced operations on it.

ADDITION

The DEFB instruction for this operation is &0F (15). It needs two numbers on the stack, and the result is returned as the Stack 'last value'. Listing 16.1 shows an example of the use of the addition operation to add 123 and 4563 together.

```
ORG     60000
LD      BC,123
CALL    11563
LD      BC,4563
CALL    11563
RST     &28
DEFB    &0F
DEFB    &38
CALL    11682
RET
```

Listing 16.1 Addition with the FPC

The two values are put on the stack by the ROM routine at 11563, then the addition is carried out by the DEFB &0F. The DEFB &38 simply tells the FPC that we've finished with it. Finally, we recover the result from the stack into the BC register pair. Run the program by typing PRINT USR 60000.

SUBTRACTION

The DEFB instruction for this is 3, and it subtracts the second value to be put on the calculator stack from the first value placed on the stack and leaves the result as the 'last value'.

MULTIPLICATION

The DEFB for this is 4, and it multiplies the last two numbers on the stack to produce the new 'last value' which is the result of the multiplication.

DIVISION

The DEFB for this operation is 5 and it works in a similar way to the subtraction operation. The first of the two numbers in the operation to be placed on the stack is divided by the second of the two numbers to go on to the stack.

There are many more FPC operations, but that is as far as I intend going in this chapter. In fact, that's as far as I intend going in this book!

I hope that you now feel confident to program your Spectrum in machine code, and trust that all your bugs and crashes will be little ones!

Appendix 1. Instructions and Op-Codes

MNEMONIC	HEXADECIMAL
ADC A, (HL)	8E
ADC A, (IX+dis)	DD 8E XX
ADC A,(IY+dis)	FD 8E xx
ADC A,A	8F
ADC A,B	88
ADC A,C	89
ADC A,D	8A
ADC A,n	CE XX
ADC A,E	8B
ADC A,H	8C
ADC A,L	8D
ADC HL,BC	ED 4A
ADC HL,DE	ED 5A
ADC HL,HL	ED 6A
ADC HL,SP	ED 7A
ADD A, (HL)	86
ADD A,(IX+dis)	DD 86XX
ADD A,(IY+dis)	FD 86XX
ADD A,A	87
ADD A,B	80
ADD A,C	81
ADD A,D	82
ADD A,n	C6 XX
ADD A,E	83
ADD A,H	84
ADD A,L	85
ADD HL,BC	09
ADD HL,DE	19
ADD HL,HL	29
ADD HL,SP	39
ADD IX,BC	DD 09
ADD IX,DE	DD 19
ADD IX,IX	DD 29
ADD IX,SP	DD 39
ADD IY,BC	FD 09
ADD IY,DE	FD 19
ADD IY,IY	FD 29
ADD IY,SP	FD 39
AND (HL)	A6
AND (IX+dis)	DD A6 XX
AND (IY+dis)	FD A6 XX
AND A	A7
AND B	A0
AND C	A1
AND D	A2
AND n	E6 XX
AND E	A3
AND H	A4
AND L	A5
BIT 0,(HL)	CB 46
BIT 0,(IX+dis)	DD CB XX 46
BIT 0,(IY+dis)	FD CB XX 46
BIT 0,A	CB 47
BIT o,B	CB 40
BIT 0,C	CB 41
BIT 0,D	CB 42
BIT 0,E	CB 43
BIT 0,H	CB 44
BIT 0,L	CB 45
BIT 1,(HL)	CB 4E
BIT 1,(IX+dis)	DD CB XX 4E
BIT 1,(IY+dis)	FD CB XX 4E
BIT 1,A	CB 4F
BIT 1,B	CB 48
BIT 1,C	CB 49
BIT 1,D	CB 4A
BIT 1,E	CB 4B
BIT 1,H	CB 4C
BIT 1,L	CB 4D
BIT 2,(HL)	CB 56
BIT 2,(IX+dis)	DD CB XX 56
BIT 2,(IY+dis)	FD CB XX 56
BIT 2,A	CB 57
BIT 2,B	CB 50
BIT 2,C	CB 51
BIT 2,D	CB 52
BIT 2,E	CB 53
BIT 2,H	CB 54
BIT 2,L	CB 55
BIT 3,(HL)	CB 5E
BIT 3,(IX+dis)	DD CB XX 5E
BIT 3,(IY+dis)	FD CB XX 5E
BIT 3,A	CB 5F
BIT 3,B	CB 58
BIT 3,C	CB 59
BIT 3,D	CB 5A
BIT 3,E	CB 5B
BIT 3,H	CB 5C
BIT 3,L	CB 5D
BIT 4,(HL)	CB 66
BIT 4,(IX+dis)	DD CB XX 66
BIT 4,(IY+dis)	FD CB XX 66
BIT 4,A	CB 67
BIT 4,B	CB 60
BIT 4,C	CB 61
BIT 4,D	CB 62
BIT 4,E	CB 63
BIT 4,H	CB 64
BIT 4,L	CB 65
BIT 5,(HL)	CB 6E
BIT 5,(IX+dis)	DD CB XX 6E
BIT 5,(IY+dis)	FD CB XX 6E
BIT 5,A	CB 6F
BIT 5,B	CB 68
BIT 5,C	CB 69
BIT 5,D	CB 6A
BIT 5,E	CB 6B
BIT 5,H	CB 6C
BIT 5,L	CB 6D
BIT 6,(HL)	CB 76
BIT 6,(IX+dis)	DD CB XX 76
BIT 6,(IY+dis)	FD CB XX 76
BIT 6,A	CB 77
BIT 6,B	CB 70
BIT 6,C	CB 71
BIT 6,D	CB 72
BIT 6,E	CB 73
BIT 6,H	CB 74
BIT 6,L	CB 75
BIT 7,(HL)	CB 7E
BIT 7,(IX+dis)	DD CB XX 7E
BIT 7,(IY+dis)	FD CB XX 7E
BIT 7,A	CB 7F
BIT 7,B	CB 78
BIT 7,C	CB 79
BIT 7,D	CB 7A
BIT 7,E	CB 7B
BIT 7,H	CB 7C
BIT 7,L	CB 7D
CALL ADDR	CD XX XX
CALL C,ADDR	DC XX XX
CALL M,ADDR	FC XX XX
CALL NC,ADDR	D4 XX XX
CALL NZ,ADDR	C4 XX XX
CALL P,ADDR	F4 XX XX
CALL PE,ADDR	EC XX XX
CALL PO,ADDR	E4 XX XX
CALL Z,ADDR	CC XX XX
CCF	3F
CP (HL)	BE
CP (IX+dis)	DD BE XX
CP (IY+dis)	FD BE XX
CP A	BF
CP B	B8
CP C	B9
CP D	BA
CP n	FE XX
CP E	BB
CP H	BC
CP L	BD
CPD	ED A9
CPDR	ED B9
CPI	ED A1
CPIR	ED B1
CPL	2F
DAA	27
DEC (HL)	35
DEC (IX+dis)	DD 35 XX
DEC (IY+dis)	FD 35 XX
DEC A	3D
DEC B	05
DEC BC	0B
DEC C	0D
DEC D	15
DEC DE	1B
DEC E	1D
DEC H	25
DEC HL	2B
DEC IX	DD 2B
DEC IY	FD 2B
DEC L	2D
DEC SP	3B
DI	F3
DJNZ,dis	10 XX
EI	FB
EX (SP) ,HL	E3
EX (SP) ,IX	DD E3
EX (SP) ,IY	FD E3
EX AF,AF'	08
EX DE,HL	EB
EXX	D9
HALT	76
IM 0	ED 46
IM 1	ED 56
IM 2	ED 5E
IN A, (C)	ED 78
IN A,port	DB XX
IN B, (C)	ED 40
IN C, (C)	ED 48
IN D, (C)	ED 50
IN E, (C)	ED 58
IN H, (C)	ED 60
IN L, (C)	ED 68
INC (HL)	34
INC (IX+dis)	DD 34 XX
INC (IY+dis)	FD 34 XX
INC A	3C
INC B	04
INC BC	03
INC C	0C
INC D	14
INC DE	13
INC E	1C
INC H	24
INC HL	23
INC IX	DD 23
INC IY	FD 23
INC L	2C
INC SP	33
IND	ED AA
INCR	ED BA
INI	ED A2
INIR	ED B2
JP (HL)	E9
JP (IX)	DD E9
JP (IY)	FD E9
JP ADDR	C3 XX XX
JP C,ADDR	DA XX XX
JP M,ADDR	FA XX XX

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

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

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

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

MNEMONIC	HEXADECIMAL	MNEMONIC	HEXADECIMAL	MNEMONIC	HEXADECIMAL
SRA (IY+dis)	FD CB XX 2E				
SRA A	CB 2F				
SRA B	CB 28				
SRA C	CB 29				
SRA D	CB 2A				
SRA E	CB 2B				
SRA H	CB 2C				
SRA L	CB 2D				
SRL (HL)	CB 3E				
SRL (IX+dis)	DD CB XX 3E				
SRL (IY+dis)	FD CB XX 3E				
SRL A	CB 3F				
SRL B	CB 38				
SRL C	CB 39				
SRL D	CB 3A				
SRL E	CB 3B				
SRL H	CB 3C				
SRL L	CB 3D				
SUB (HL)	96				
SUB (IX+dis)	DD 96 XX				
SUB (IY+dis)	FD 96 XX				
SUB A	97				
SUB B	90				
SUB C	91				
SUB D	92				
SUB E	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		8-bit subtract with carry
X0R 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 subtractionusing 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.

Index

The **Spectrum+2** is a machine long dreamt of by home computer users — 128K of memory, enhanced sound, built-in data recorder and joystick port, and much more. And here, in **Spectrum+2 Machine Language for the Absolute Beginner,** is the book which shows you how to utilise to the full the +2's extraordinary potential.

From the introductory chapters, which lay down the first principles of machine code and its differences to and advantages over BASIC, Joe Pritchard conducts you through a comprehensive course in Z80 programming. Registers, addressing modes, eight and 16-bit operation, jumps, loops and interrupts, keyboard reading, screen handling and graphics are dealt with in turn. The course concludes with some useful ROM routines and appendices for quick reference.

Those already fluent in machine language will also find much that is relevant, as Joe Pritchard examines those features of the Spectrum+2 unfamiliar to even experienced Sinclair users. For instance the chapter on Sound is alone a most worthwhile addition to any programmer's library.

Spectrum+2 Machine Language for the Absolute Beginner — the perfect companion to a powerful new computer.

£8.95

Melbourne House Publishers

ISBN 0-86161-209-4

9 780861 612093