SPECTRUM MACHINE CODE MADE EASY

Volume One

For Beginners

James Walsh

SPECTRUM MACHINE CODE MADE EASY

Volume One

For Beginners

James Walsh

INTERFACE PUBLICATIONS

Then at the balance let's be mute,
We never can adjust it;
What's done we partly may compute,
But know not what's resisted.

Robert Burns

DEDICATION

To Margy, with love
From
Us

To Charles and Emma — THANKS for all the help

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

ISBN 0 907563 43 0

Cover Illustrator, Stuart Hughes

Typeset and Printed in England by Commercial Colour Press, London E7.

THE AUTHOR

James Walsh, at the time of writing this book, is sixteen years old and attends Davenant Foundation School in Loughton, Essex.

He has already established a reputation as a regular contributor to several nationally distributed computer magazines, and has previously contributed to two published books on Sinclair computers.

In addition to writing articles, software reviews, and books, he finds the time to be a keen photographer, a mediocre musician and, for the most part, a worry to his teachers.

He hopes to publish his next book in the not too far distant future — exams permitting — although his more immediate ambition is to travel abroad.

The time given to writing this book has done little for his social life — something he now hopes to find the time to correct. In the interim he acknowledges both the patience and support of all his family and friends.

Clive C. Walsh

CONTENTS

INTRODUCTION

The aim of this book is to provide a coherent and digestible introduction to Machine Code programming — deemed necessary on the basis that the author wished there had been one available when he first started to struggle with this somewhat mysterious topic. It was only later that he realised that there really is no mystery.

Every effort has been made to avoid falling into the trap of either assuming "pre-knowledge" on the part of the reader, or, indeed, of presuming him to be an ignorant fool! It has therefore been necessary to approach the subject from the point of view of the "learner", attempting always to anticipate the problems that he or she might encounter. If you find the result too easy — this book is probably not for you. If you find it too difficult — I apologise, but must presume you to be beyond help (try becoming a Member of Parliament instead).

Computers Don't Speak No English!

Chapter 1
WHY MACHINE CODE?

The answer to this question is, quite simply — "Machine Code is the computer's own language". To that extent therefore, "It makes better sense to the computer". I could offer you a much more complex and technically correct definition of Machine Code, and doubtless leave you somewhat less well-informed than when you started. However, this book represents an attempt to remove the mystery and confusion that surrounds the business of communicating with computers and in particular, with your Spectrum. Therefore, every effort has been made to present information and explanations in the simplest way possible. If, by the time you have worked your way through the whole of the book, you come to have a firm grasp of the essentials of Machine Code and are confident enough to be able to use it, then the effort will have been worth while.

The best that computers can manage is to add 'one and one' and reliably arrive at "TWO" as the correct answer. That is neither particularly clever, nor awesome. Nonetheless, they do seem to have an infinite capacity for intimidating human beings, so that it is possible to be easily convinced that computers are too complicated, too quick, too clever, or just too frightening to get to grips with. Despite any appearances to the contrary, your Spectrum is nothing more than a combination of electronic components which, collectively, produce a capacity to process the information that it's given, according to the instructions that it's given. Albeit that the end-result is a sophisticated piece of equipment, it is a tool to be used, but a tool that has no mind or will of its own. It can only deal with what it is given, and can only do what it does because 'you' tell it what it is to do. It does carry out its task most efficiently and reliably and can therefore handle lengthy complicated calculations in a fraction of the time taken by the human brain. However, as already said, its real claim to fame is its ability consistently to get the right answer when asked to 'add one and one'.

Why go to all the trouble to understand the nature of machine code and then spend time trying to learn how to use it? Do you not already have a perfectly adequate 'language' with which to communicate with your Spectrum — BASIC? Certainly, BASIC is a language with which you will by now have become familiar, and it is a language which at least appears reasonably intelligible. In fact, that is precisely what it is — a computer language that is written for humans. Its purpose is to allow us all to 'talk' to our ZX80/ZX81/Spectrum in a compact and orderly fashion, using a language-form which is understandable to us, and can also be 'interpreted' by the computer. Yes, 'interpreted'. Within the computer's memory there is a complete BASIC to Machine Code dictionary. The computer therefore relies on this totally to interpret and so understand anything that is fed into it in BASIC, before it can carry out the instructions given.

If, therefore, the aim is to achieve easier understanding of and a quicker response from your Spectrum, being able to 'speak its language' starts to become more important. It is easy to envisage just how difficult it would be to communicate if, when travelling abroad in a foreign country, you met up with someone who spoke only their own 'foreign' language, and you knew nothing of it whatsoever. Without resorting to some rather clever sign-language it would be virtually impossible to make much progress, and in fact, the sign-language would only be a joint attempt to find a common simple language that each could understand. If the services of an interpreter could be enlisted then the process would start to become a little easier. However, everything passing between you would have to be interpreted and then re-interpreted, and so the

'conversation' would inevitably be rather limited, and very slow.

This then is the situation in the Spectrum — it has an in-built interpreter. Whatever is entered in BASIC is interpreted into its own language, and vice versa, and there is therefore a natural limit to the extent to which you can communicate with it, and also the speed at which that communication can take place. When programs are written in BASIC, before operating them the Spectrum must first store and then interpret all commands and data. The 'interpretation' into Machine Code is in order to allow the Z80 micro-processor to function — this being the ONLY language that it is capable of understanding. Inevitably, this process of searching the computer's storage facility, bringing out instructions and data and then executing them, takes times. If, in addition, everything must also be interpreted from BASIC to Machine Code before any of this can even start, then the time taken is that much greater. There are times when this can be an important consideration, particularly for instance, where control of graphics or speed of program reaction are involved.

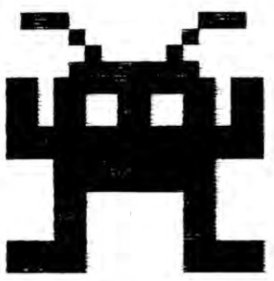
As when travelling abroad, if ultimately you wish to be able to communicate speedily, easily and accurately, then there is no substitute for learning the language of the country in which you are travelling. You have chosen to travel in the computer's own territory, and you must therefore learn its language. It might seem unreasonable, but take it from me that your Spectrum is not going to learn yours — not just yet anyhow!

Of course learning a new 'foreign' language is not necessarily all that easy. Fortunately, computer language is extremely simple, though how the computer then goes on to use it is a different story, but one which we will be going through together. The computer is good at adding 'one and one' and in that this is really all that it can do, it spends its whole existence dealing in 'noughts' and 'ones'. It is possibly useful to mention at this stage that, throughout this book, the convention of signifying zero by Ø will be adhered to in order to avoid confusion with the letter 'O'. To understand why the computer should only deal in noughts and ones, and how this simplistic basis can enable it to perform the most complex of tasks, it is necessary to have some understanding of the way in which the computer is put together, and how it then manages to work. Before doing so, let us just take a look at the speed at which it can respond to instructions given in Machine Code, rather than BASIC.

Type in this program:

```
   5 BRIGHT 1: PAPER 4: BORDER 6
: INK 1: CLS
  10 PRINT AT 5,8;"██        ██"'"
  █  █"'"  ████████"
  15 PRINT AT 10,0;"
██ ██ ███"'"  ████████████"
  20 PRINT AT 13,0;"
██████"'"
  ██    ██████"'"
  25 PRINT AT 0,0;"Now storing i
mage"
  30 FOR x=0 TO 7167
  40 POKE (25600+x),PEEK (16384+
x)
  50 NEXT x
  60 PRINT AT 0,0;"Transfer comp
lete"
  70 PAUSE 30
  80 CLS
  90 PRINT "Now transfering it b
ack"
 100 FOR x=0 TO 7167
 110 POKE (16384+x),PEEK (25600+
x)
 120 NEXT x
 130 PRINT "END OF PROGRAM"
```

```
Now storing image
```



Slow isn't it? Now type in the equivalent Machine Code version of the program as set out below, and see how it compares. Delete Lines 3Ø, 4Ø, 5Ø, 7Ø, 1ØØ, 11Ø, 12Ø, add these:

```
 150 STOP
 200 FOR a=25501 TO 25524: READ
a$
 210 LET ans=(CODE a$(1)-48)*16
 220 IF ans>9*16 THEN LET ans=an
s-7*16
 230 LET l=CODE a$(2)-48: IF l>9
 THEN LET l=l-7
 240>LET ans=ans+l: POKE a,ans
 250 NEXT a
 260 RETURN
 300 DATA "21","00","40","11","0
0","64","01","00","1C","ED","B0"
,"C9"
 310 DATA "21","00","64","11","0
0","40","01","00","1C","ED","B0"
,"C9"

  30>RANDOMIZE USR 25501

 100 RANDOMIZE USR 25513

  80 PAUSE 100

RUN
```

Now add this Line:

```
1000 CLS : PAUSE 30: RANDOMIZE U
SR 25513: PAUSE 30: GO TO 1000
```

. . . . and execute.

```
RUN 1000
```

Now are you convinced?

To return to the manner in which the computer operates. Essentially, it performs calculations arithmetically, using a binary system of arithmetic and a memory in which it stores information. Your Spectrum, as a digital computer, recognises only TWO states; it is able to recognise either the presence or the absence of an electrical impulse. It operates therefore on the basis of being either 'on' or 'off'', and hence the term 'binary system' and the computer's 'bi-stable' nature. Given this importance piece of information, we can take the next step of representing these two states by the use of 'noughts' and 'ones'. It is a logical progression to represent 'no electrical signal' as 'Ø', and 'some electrical signal' as '1'. The computer's two stable states can therefore be represented as:

1. Is there a signal? Yes — represent as '1'
2. Is there a signal? No — represent as 'Ø'

The easiest comparison to make is that of the ordinary switch, which can be in either the 'ON' or the 'OFF' position. When the switch is on, electrical current is able to flow — when it is off, no current can flow. It might sound as though we have now diminished the computer from something that can at least count up to two, and left it as something more akin to a simple household light switch. To an extent this might indeed be true, but clearly would dramatically under-sell the poor old computer. In fact, your Spectrum — or rather its Z80 micro-processor — has the capacity to string together a series of such ON/OFF decisions and so produce a large number of discretely different combinations, each of which has its own

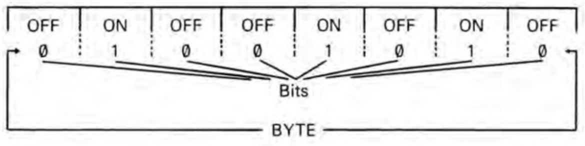
particular significance. To understand that more fully take, for instance, just two such simple ON/OFF 'switches' and combine them. This then allows for four discretely different states:

1. ON and ON	becomes 11
2. ON and OFF.	becomes 1Ø
3. OFF and ON	becomes Ø1
4. OFF and OFF.	becomes ØØ

Combining three such 'switches', which are still simply representative of the ON/OFF states recognised by the computer, allows for eight different specific states:

1. OFF and OFF and OFF.	ØØØ
2. OFF and OFF and ON	ØØ1
3. OFF and ON and ON	Ø11
4. OFF and ON and OFF.	Ø1Ø
5. ON and ON and ON	111
6. ON and ON and OFF.	11Ø
7. ON and OFF and OFF.	1ØØ
8. ON and OFF and ON	1Ø1

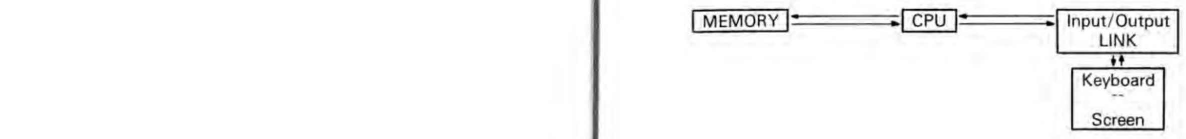
What should by now be starting to become apparent is the simple rule of 'combinations'. If we take two specific states they can be combined in a maximum of 2 × 2 (i.e. four) different ways. Take three as the starting point and a maximum of 2 × 2 × 2 (i.e. eight) discretely different combinations become possible. Now, just as we are able to string letters of the alphabet together, and so make specific 'combinations' which we recognise as words, each with its own particular meaning, so can the computer string together combinations of the two digits 'Ø' and '1' — and so produce specifically identifiable 'words'. Your Spectrum can handle 'words' made up of eight digits, and using the rule of combinations it can be seen that this provides for a 'vocabulary' of 256 possible different 'words' — 2 × 2 × 2 × 2 × 2 × 2 × 2 × 2, or 'two to the power eight'. In stringing the two digits together in this fashion, 'words' are created which the computer is capable of understanding — each 'word' being known as a 'binary number'. The name given to such a binary number is a BYTE — the digits of which it is made are referred to as BITS.



In this introductory chapter an attempt has been made to help you, the reader, to at least start to come to terms with both the simplicity and sophistication of your computer. Machine Code is the language that it uses, and therefore understands, and you have come to see that this can then be reduced to strings of Ø's and 1's. The next chapter will focus on how the various components are organised in the computer as a basis, at least in outline, for understanding something of what it uses in order to function, and so how the 'binary arithmetic' is put to such effective use. The next task will then be to look at how instructions are 'packaged' before being given to the computer in words that it can understand, and we can handle. Whilst it will be necessary to deal with the issue of how 'we' interpret a binary number for ourselves, you need not resign yourself to the dreadful prospect of having to read unending sequences of Ø's and 1's, or of having to feed your computer with such a diet.

If Manners Maketh Man, Numbers Make The Computer......

Chapter 2
WORDS AS NUMBERS



Above is a simple diagrammatic representation of how the Spectrum is organised. Reference has already been made to the Z80 micro-processor, and its function to SEARCH, LOCATE, and EXECUTE. This, the Central Processing Unit (CPU), as its name implies, is the central component of the computer. It is in two-way communication with the other two components — the LINK and the MEMORY.

The Link: allows the CPU to output the results of its labours to other devices — such as a television/VDU screen — in order to display visually those results. Alterniatively, it can allow them to be placed in permanent storage on an audio cassette in a tape recorder, or a more refined storage medium e.g. in disc form. The Link is also capable of permitting information to be inserted (INPUT) from outside the computer — from the keyboard for instance, or from an external store. For example it can use this to link into and operate circuits containing sensors, and so allow it to do anything from controlling production in a factory, to interrupting your game of Space Invaders in order to let you know that your dinner has just burned!!! Hence it is known as the computer's INPUT/OUTPUT facility.

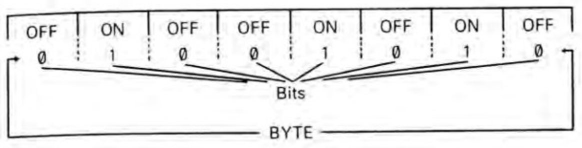
The other component is the MEMORY. This does exactly as you might expect — information and data is stored, and can be recalled. Because it reliably remembers whatever is put in, it is to the Memory therefore that the CPU turns when needing to find out what is expected of it. There are essentially two types of Memory — RAM and ROM.

RAM is short for Random Access Memory, which means that whether you wish to read or write to the first or last part of memory, it takes precisely the same amount of time. This is in direct contrast to other storage devices, such as a cassette tape or record disc where, in order to find the last piece of information it is necessary first to read through all the preceding information. Similarly, with an audio tape, it is possible to write into (put into memory) RAM, as well as to read out.

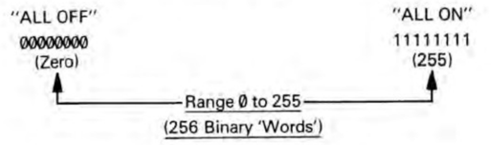
In contrast, ROM stands for Read Only Memory. As you might expect, this indicates that it is only possible to read what is contained in that part of the Memory, without the facility of changing it in any way. This is similar to the way in which it is possible to read (play) the information (music) that is stored on a record disc, whilst being unable to do anything about adding fresh information (except by an act of sheer vandalism!!). You might try of course, but, just as if you were to press the 'record' button of a tape recorder without there being a tape inserted, the cogs would go round, but nothing would happen. For the purpose of clarification it does need to be said that ROM is also random access memory in precisely the same way as RAM — the real difference is that ROM is more accurately described as Random Access Read Only Memory.

So far I have done my best to avoid the more traditional 'numbers approach' to computers and computing, because I remain hopeful of retaining your interest. Nonetheless, numbers do have their rightful place in the scheme of things and so cannot be ignored. Focussing now on how numbers are stored, and used, will provide the essential basis for understanding how to use and communicate with your

Spectrum as an integrated whole not just as an assortment of separate components. Referring back to the analogy of ON/OFF switches:



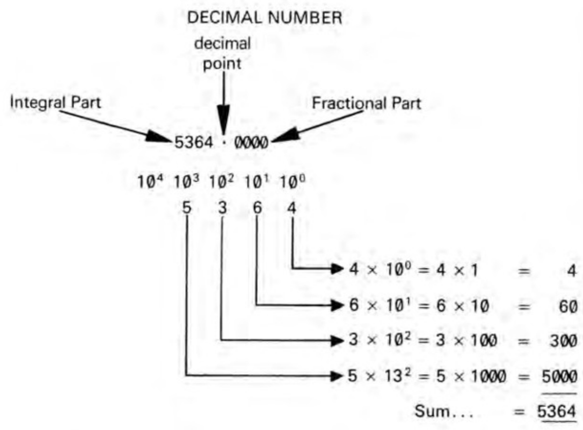
It can be seen that binary number Ø1ØØ1Ø1Ø is a specific 'binary word' that falls somewhere within the range of Spectrum's vocabulary of 256 words. The extremes of that range are represented by "All Off" and "All On".



Why, though, should a string of eight noughts be "zero", and eight ones be equivalent to the decimal number 255? The answer lies in the type of arithmetic used by the computer — BINARY ARITHMETIC — which is perhaps best understood by first exploring the structure of the 'decimal' system with which we are most familiar.

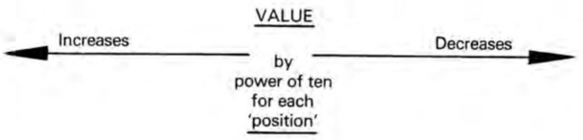
The Decimal System is based on ten digits (Ø to 9), and can therefore equally well be referred to as "Base Ten". It also employs a system of 'Positional Notation'. This allows the value of any digit in a number to be calculated on the basis of its position in relation to the Decimal Point. As a matter of accuracy, positional or point notation relies upon a digit's position in relation to the Unit Point — it is not "the decimal point" as such. In binary arithmetic for example, it becomes the

Binary Point. The Unit Point is that point which separates the Integral part from the Fractional part of a number. Staying with the decimal system for the moment, the value of a digit in a number is the product of itself and the 'power of 10', determined by its position in relation to the decimal point. The digits in the number are arranged in 'columns', each column having a value ten times that of the column immediately to its right.

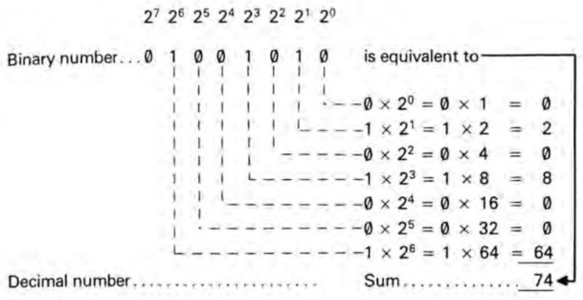


DECIMAL NUMBER
decimal point
Integral Part
Fractional Part
5364 · 0000
10^4 10^3 10^2 10^1 10^0
5 3 6 4
$4 \times 10^0 = 4 \times 1 = 4$
$6 \times 10^1 = 6 \times 10 = 60$
$3 \times 10^2 = 3 \times 100 = 300$
$5 \times 13^2 = 5 \times 1000 = 5000$
Sum... = 5364

Each digit/column has a 'power' ten times greater than that of the digit/column immediately to its right.

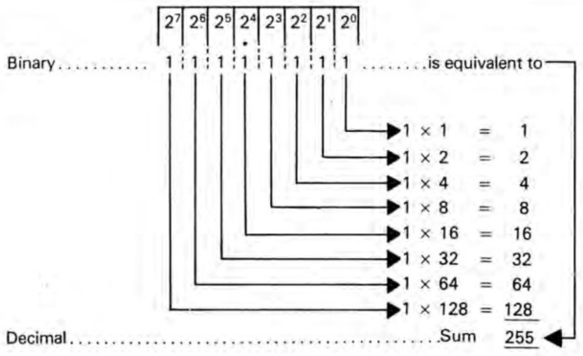


The Binary System — is based on only two digits (0 and 1), and is therefore to BASE TWO. As for Decimal, a system of positional notation is employed, though each digit in a binary number now has a value which is to a 'power of two' greater than that to its immediate right.



2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
Binary number... 0 1 0 0 1 0 1 0
is equivalent to
$0 \times 2^0 = 0 \times 1 = 0$
$1 \times 2^1 = 1 \times 2 = 2$
$0 \times 2^2 = 0 \times 4 = 0$
$1 \times 2^3 = 1 \times 8 = 8$
$0 \times 2^4 = 0 \times 16 = 0$
$0 \times 2^5 = 0 \times 32 = 0$
$1 \times 2^6 = 1 \times 64 = 64$
Decimal number.......................
Sum.............. 74

It can now be seen that in the example used above 01001010 is the binary equivalent of Decimal 74. Similarly, 00000000 = Zero (Decimal), and 11111111 = 255 (Decimal).

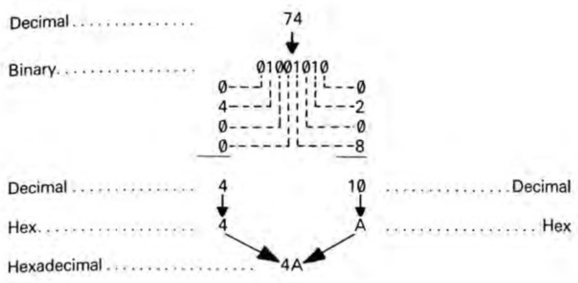


Whilst the computer uses the binary equivalent of the decimal number on which to work, you were promised in the previous chapter that it would not be necessary for you to remember and use strings of Ø's and 1's in order to be able to communicate with your Spectrum. Sadly, neither can you just rely on the familiar decimal numbers involved. However, whilst "Base 1Ø" is of little use, and "base 2" is impractical, "Base 16" happens to be uniquely convenient. Do not despair — it really is easier than it sounds. Applying the principles already outlined, Base 16 — or HEXADECIMAL — employs sixteen digits. The first ten in the sequence correspond to those in the decimal system (Ø to 9), the remaining six being represented by the first six letters of the alphabet — A, B, C, D, E, F. The table below sets out the Binary/Decimal/Hexadecimal relationships.

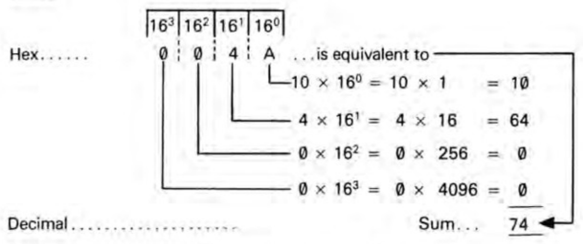
Binary Number	*Decimal Equivalent*	*Hexadecimal Equivalent*
ØØØØ	Ø	Ø
ØØØ1	1	1
ØØ1Ø	2	2
ØØ11	3	3
Ø1ØØ	4	4
Ø1Ø1	5	5
Ø11Ø	6	6
Ø111	7	7
1ØØØ	8	8
1ØØ1	9	9
1Ø1Ø	1Ø	A
1Ø11	11	B
11ØØ	12	C
11Ø1	13	D
111Ø	14	E
1111	15	F

A full table of the Binary/Decimal/Hexadecimal equivalents from Ø to 255 can be found at Appendix A.

Let us take again the earlier example of 74 (Decimal), and its equivalent of Ø1ØØ1Ø1Ø (Binary) and note the convenient way in which it converts to Hexadecimal (frequently shortened to Hex for convenience):

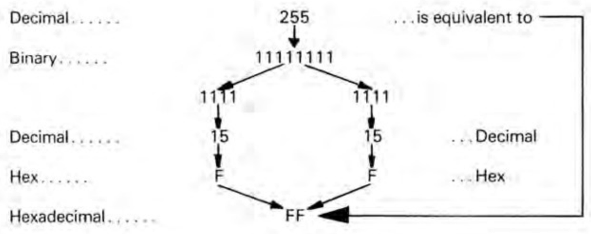


Check:



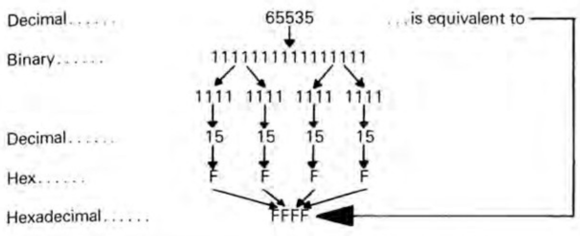
Binary Number	*Decimal Equivalent*	*Hexadecimal Equivalent*
Ø1ØØ1Ø1Ø	74	4A

If we do the same for 255, as the highest number to be handled by the CPU, the ease with which any 8-bit binary number can be represented by a 2 digit Hex number:



Binary Number	*Decimal Equivalent*	*Hexadecimal Equivalent*
11111111	255	FF

This 'grouping' of binary digits into 4's, and then combining their Hex equivalents, becomes even more important as the most convenient method of representing ADDRESSES or MEMORY LOCATIONS. Great stress has so far been placed on the fact that 255 is the highest number that can be handled. However, this restriction cannot be accepted, and shortly I will demonstrate how your Spectrum's CPU is able to utilise its ability to handle 255 binary words IN COMBINATION with its ability to add 1 and 1, and so arrive at 65536!!!! Dealing with a range of numbers from Ø to 65535 requires 16-bit binary numbers, the highest of which will be 1111111111111111 — the binary equivalent of 65535 (check it out if in doubt). All the numbers involved — whether in binary or decimal — are now becoming unmanageably large. The particular value of Hexadecimal will now become more apparent — not least because it allows for any binary number to be rearranged into groups of four digits, each group then to be represented by a one-digit Hex number, and these then combined to give a four-digit Hex equivalent of any 16-bit binary number:



Binary Number	*Decimal Equivalent*	*Hexadecimal Equivalent*
1111111111111111	65535	FFFF

Every number in the range Ø to 65535 can now be reduced to a Hex number of four digits in the range ØØØØ to FFFF.

As with all 'numbering systems', arithmetic calculations can be performed in Binary Arithmetic and, of course, the computer is reliably adept at doing just that. Everything is handled on the basis of either Addition or Subtraction — your Spectrum even manages to do its subtractions by 'Addition', but more of that shortly. The advantage of course is that we at least do not have to get over-concerned about Multiplication and Division functions. In general terms, conventions (or rules) that apply in decimal arithmetic apply equally well in binary arithmetic. Thus, the highest number that can occur in any column is 1; that is, one less than the total number of digits allowed for in the system (two minus one). In decimal the highest digit is 1Ø – 1 = 9. When performing additions between two binary numbers the familiar rule of "carry one to the next column" applies when the sum of two digits is greater than 1. Reference to the table below will clarify the basic "truths":

```
0+0   = 0
0+1   = 1
1+0   = 1
1+1   = 10 ("nought, carry one")
```

+	0	1
0	0	1
1	1	10

An example of the addition of two binary numbers would appear as follows:

```
              01001010        (Decimal  74)
            + 00111010        (Decimal  58)+
Carry......   0111010-                     =
Sum......     10000100        (Decimal 132)
```

Applying the same principles to Subtraction (except that this time we borrow two, rather than carry one), is essentially a reversal of the process of addition:

```
              10000100        (Decimal 132)
            - 00111010        (Decimal  58)-
Borrow......  01111010-                    =
              01001010        (Decimal  74)
```

This is in fact a rather more complex process than it need be, and subtractions are best carried out using a method known as "The Two's Complement" — this is what the computer does in reality. In the above example the result of the subtraction can more easily be achieved by ADDING the "two's complement" of 00111010 (the Subtrahend) to 10000100 (the Minuend). To determine the "two's complement" of any binary number simply change all the 0's to 1's, and all the 1's to 0's, then add 1 to the result:

```
Subtrahend......            00111010  (Decimal  58)
Two's Complement......      11000101 + 00000001
 ┌─────────────────────── = 11000110
 │
 │ Minuend......            10000100  (Decimal 132)
 │
 │                        +
 └►Two's Complement......    11000110
        Sum             ──► 101001010
                             ¦└──┬───┘
    "Sign Bit"                   │
                           Decimal  74
```

The eight least significant bits are the same as before (Decimal 74), though there is now one additional "Carry Bit" — otherwise known as a Sign Bit. This can be disregarded in terms of determining the numerical result, but is taken as the indicator of whether the result of the calculation is positive or negative. When the Carry Bit is 1 it signifies that the resultant number is positive in value. Conversely, when the Carry Bit is 0, the result of the calculation has a negative value.

Computers Do It In Tower Blocks

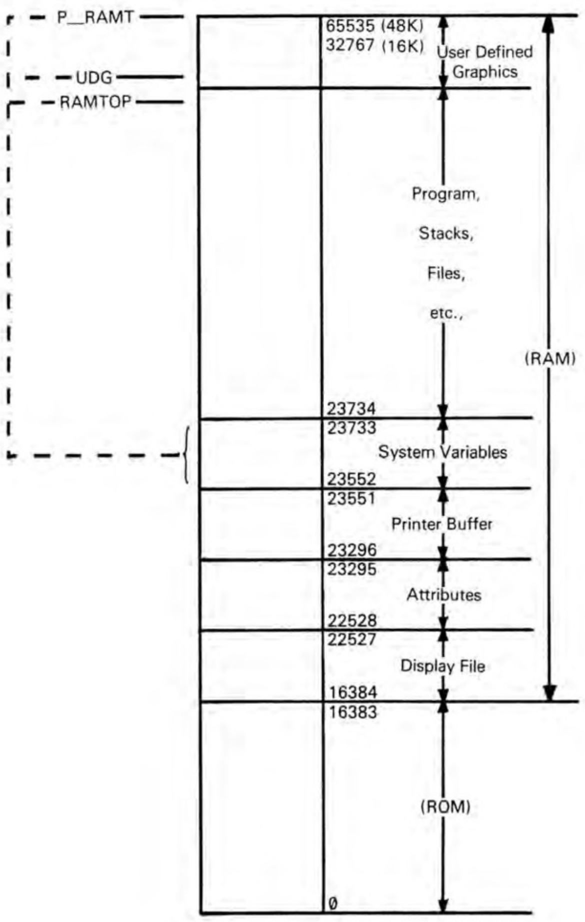
Chapter 3
ADDRESSES AND HOW TO GET THERE

It is worthwhile summarising the ground covered so far. We started from the very generalised position of describing machine code as the computer's own language, and considered the benefits that can come from learning that language. Similarly, starting with a very generalised view of how your Spectrum works, we have started to look in more detail at how its various components link together and make use of binary arithmetic in order to function. This has necessitated coming to grips with another numbering system — Hexadecimal — in order to learn how to be able to reduce 8-bit binary numbers to 2-bit Hex, or 16-bit binary numbers to 4-bit Hex numbers.

Having given particular weight to Spectrum's limited vocabularly of 256 'words'. I have given notice of its ability to store information in over 65000 separate locations. This system of "Addressing" will come in for considerable scrutiny during the course of this chapter. In addition we will look in much more detail at how Spectrum's Memory is structured and organised, and start the process of learning how to manipulate the contents of that memory.

Our first task is to explore the structure and organisation of Spectrum's Memory by referring to the Memory Map (see over).

Each location in memory is a place where information can be stored, and is commonly known as an ADDRESS. This



terminology has not been arrived at by chance but is the sensible use of a word and concept familiar to us all. Each location is allocated a number in the range of 0 to 65535 — the number thus becoming that location's specific address. A glance at the Memory Map above reveals that the first 16384 (or 16K) addresses are given over to ROM. This, as we now know, is Read Only Memory, and is essentially a block of 16384 locations where information is stored at the time of the ROM-Chip's manufacture. It is therefore NOT amenable to change or alteration by the user. In the 16K version of the Spectrum there are, in addition, a further 16384 RAM locations. The 48K version of this machine uses the maximum number of locations that can be addresses by the Z80 CPU; therefore, in addition to the 16K of ROM there is a further 48K of RAM (locations 32767 to 65535 — or, if expressed in Hex, 8000 to FFFF). The combination of 16K ROM and 48K RAM gives a total of 64K of memory — or 65536 bytes.

Before going further I had best lay to rest any concern or confusion that remains in relation to this number "65535". By now it must be clear that the Spectrum's CPU can handle 256 different 8-bit bytes. Consider now, hypothetically, its memory to be rather like an enormous tower-block of flats — 256 storeys high, with each floor consisting of 256 separate dwellings. Each separate 'flat' could be allocated an address based on the 'floor' on which it is situated and its position on that particular floor. Taking the floors as being numbered sequentially 0 to 255, and the dwellings (locations) similarly numbered 0 to 255 on each of the 256 floors, it becomes possible to locate any particular dwelling by specifying its 'floor' and 'door' numbers. For example; Floor 10, Door 54. The location of that address within the total 65536 dwellings could then readily be determined as follows:

(Floor number × 256) + Door number.
(10 × 256) + 54 = 2614.

Similarly, the highest location would be (255 × 256) + 255 = 65535. This, in effect, is how your Spectrum sets about addressing its 65536 memory locations — though, perhaps predictably for a Sinclair product — though really a function of the Z80 — it insists on doing it "upside down". It uses two bytes of memory to handle an address; the first to store the 'least significant' part of the address (54 in the example above), and the second to store the 'most significant' part (10 in the example given). The 16-bit address codes used by the Z80 CUP are therefore two 8-bit bytes regarded as one long number; the first eight bits of which constitute the Least Significant Byte, and the last eight the Most Significant Byte — generally referred to as the contractions "LSB" and "MSB".

This can be exemplified (and put to good use) by a simple formula for determining how many bytes of memory have been used to store a Basic program. Key in, or load any Basic program then, find the bytes used as follows:

```
PRINT (PEEK 23653 + (PEEK 23654) * 256) – (PEEK
23635 + (PEEK 23636) * 256)
```

This simply identifies the address at which free memory starts (STKEND) and subtracts the start address of the Basic program. These addresses are found by PEEKing the System Variables given above — each address being stored in two bytes of memory, the LSB first. The first address is found therefore by 'looking into' memory location 23653, extracting its contents (one 8 bit-byte) and adding it to the contents of the next memory location multiplied by 256. The second address is found in a similar fashion.

In exploring addresses I have slipped in the word PEEK, which is totally descriptive of the function. It allows us to peek (or peep) into a memory location and see what is in there — rather like looking in through the window of one of the 65536 flats in our over-grown tower block. In that part of memory given over to ROM this is all that we can do — LOOK but not TOUCH! With the RAM component of memory however we can both look and touch. The command for 'touching', or changing, is "POKE", and this allows the user to change the contents of a

memory location by POKEing in a new value. Try it! Ask your Spectrum to PRINT PEEK n, where "n" is a RAM location of your choice. Once the contents of that location have been identified (printed on the screen) POKE n,x — where "n" is the same location and "x" is a different value of your choice, in the range Ø to 255. Now PRINT PEEK n again and confirm that you have successfully changed the contents of that memory location. Lots of practice of PEEKing and POKEing will come your way as we progress through the book!!

Maybe without fully realising it, we have arrived at the stage of being nearly ready to start talking to the Spectrum in its own language. We know how the 'words' are made up, where they are stored, and how in principle, we get them into the computer — we poke 'em in! What other basics do we need to know? Certainly we must be able to prepare the computer to receive our instructions, and of course it will be necessary to ascertain precisely which words can be understood and processed. Strictly the Z80 microprocessor should be able to understand just 256 m/c instructions, but again, not surprisingly, it manages to rather stretch things, and so, by judicious use of supplementary "prefixes", in excess of 6ØØ can be recognised and executed by the CPU. All are listed at Appendix B. Now for a few basic principles.

Machine Code routines can always be called into action via a BASIC program by use of the command "RAND USR". So, to gain access to a machine code routine employ the User Sub-Routine, specifying where in memory the routine is to be found. For instance, RAND USR 3ØØØØ would call in a machine routine, the first byte of which was stored at memory location 3ØØØØ — the remainder of the routine being stored sequentially from "there on up". How is the start location determined in the first place? Well, we just make up our own mind and dictate to the computer. Entering CLEAR 29999 (or any other convenient location) has the effect of setting RAMTOP at that address, and thus identifies it as the last location at which any part of a BASIC program can be inserted. The machine code routine then starts at the next location — 3ØØØØ. Using the

NEW command clears out RAM only as far as RAMTOP, and any machine code stored above RAMTOP is protected from being either NEWed or over-written by the BASIC. Using the command RUN will CLEAR the variables, but without changing RAMTOP in any way. Therefore, using CLEAR with a specified memory address is a means of moving RAMTOP 'up' so as to make more room for BASIC (by over-writing the User-Defined Graphics), or 'down' to increase the area of RAM that is protected from NEW.

Having decided where the machine code is to be stored, now how is this achieved? As mentioned earlier, it must be stored sequentially 'cos that's how the Spectrum's CPU goes about its business. The FIND and EXECUTE process starts at the address stipulated then works its way through each successive location in memory — interpreting everything that is found as an instruction and so sets about executing it. This process continues until instructed to stop — or until it has a brain-storm. An enjoyable program that demonstrates this 'sequential' manner of working is given below. Please do not concern yourself about how it works — it relies wholly on the fact that the Spectrum ROM has become delirious!!!!

```
   5 RESTORE
  10 READ n
  20 LET a=120
  25 PLOT 55,27: DRAW a,a,n*PI
  32 CLS
  33 IF n=9999 THEN GO TO 5
  35 GO TO 10
  40 STOP
  50 DATA 597,631,315,343,297,63
1,613,787,313,741,187,147,279,99
99
```

That diversion over, let us now look at four options of LOADing-in machine code:

1. POKE one byte at a time.

   POKE X,Y

Where X is the start address and Y is the value to be entered.

The second byte is POKEd at location X + 1, the third at X + 2 etc., etc.

2. FOR/NEXT Loop.

```
10 FOR A = 1 TO X
20 INPUT Y
30 POKE A + start location, Y
40 NEXT A
```

3. Data READ.

```
10 FOR A = 1 TO X
20 READ N
30 POKE start location + A,N
40 NEXT A
50 DATA N1,N2,N3,N4...
```

Where N1, N2, N3, N4...are the sequential bytes of the machine code routine.

4. LOADing from tape.

```
CLEAR chosen location
LOAD "name" CODE.
```

Options 2 and 3 offer two different bases for a Machine Code Loader. Clearly, loading-in binary combinations is definitely out, and our newly-acquired knowledge of Hexadecimal can be put to good use. Hey Presto! — one HEX. LOADER:

```
  10 PRINT "       ***Machine Code
Loader***"
  20 PRINT "          © James Walsh
 1983"
  30 INPUT ''"Start Address?";ad
dress
  40 PRINT '''"Enter code one by
te at a time"
  50 PRINT "when prompted. In UP
PER CASE only"
  60 PRINT address;" = ";
  70 INPUT a$
  80 IF a$="END" THEN GO TO 1000
  85 IF LEN a$<>2 THEN GO TO 70
  90 LET hi=CODE a$(1)-48
 100 IF hi>9 THEN LET hi=hi-7
 110 LET ans=16*hi
 120 LET low=CODE a$(2)-46
 130 IF low>9 THEN LET low=low-7
 140 LET ans=ans+low
 150 PRINT a$
 160 POKE address,ans
 170 LET address=address+1
 180 GO TO 60
1000 PRINT AT 21,0;"Program comp
lete"
```

Some Registers Aren't For Cash

Chapter 4
WORKING ON THE REGISTERS

Having read the previous chapters, you should have a reasonable idea of the range of numbers which the computer can understand, the way in which we can use these numbers, and the way in which the computer uses them to make it possible for us to manipulate the memory of the computer; as if it were a long row of empty boxes — all of which have particular numbers so that we can gain access to those boxes without having to look through each one — and then add to and subtract from them at a later date. What we shall be doing in this chapter is looking at the way in which the computer handles these numbers when they are out of their boxes, and how we can manipulate them. We shall look at the way in which the computer holds these numbers, the way in which the computer holds the commands in machine code, and we shall also look at some of the simple machine code instructions for this purpose. We shall then start exploring how we can use these instructions in our own programs. You will then be left with a problem to solve using a machine code program.

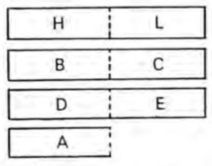
REGISTERS

It is all well and good being able to put numbers in boxes and take them out again, but really it is not much use unless it is then possible to do rather more than just that. When you are writing a BASIC program you use a variable. For example, if you want a number to be stored as variable A you would simply LET A= the number. You would then be able to do whatever you wished with that variable A. You could add it to another number, double it or subtract it, and when finished you would be able to clear that variable back to nought simply by typing LET A=Ø.

Unfortunately there are no variables in machine code. Instead we have what are known as REGISTERS. There are seven single-byte registers which can easily be used and manipulated. There are others, but these are mainly used by the computer. We can indeed utilise them, but with a little more difficulty and for this reason we shall leave these registers until later. For the vast majority of the time you will spend machine code programming, the only registers used will be the "easy" seven. Each register has a letter of identification: A, B, C, D, E, H, L. There is no real reason why these seven letters were chosen, so there is no point in worrying yourself by trying to make particular sense of the convention. As you now know, the computer, when handling a single byte, can only deal with a number of Ø to 255. Because each of these registers is a SINGLE BYTE REGISTER, 255 is the maximum number which can be stored in any one register. You have doubtless already noted that this is really too low a number to be of practical value — how often do you yourself actually manage to write a program in which you have no numbers above 255? To remedy this, what the computer does is to group two single-byte registers together to make what is known as a REGISTER PAIR. If two registers are combined together in this manner, into one register pair, the maximum number which together they can handle is 256 multiplied by 256, minus 1, which makes a grand total of 65535. In computer terminology this range of Ø to 65535 is represented as 64K. The reason that we have to minus 1 at the end is purely because the range is in fact nought to 65535 which means that you have to add an extra 1 byte to record the number zero.

Registers cannot be combined together in any random

combination. The fixed combinations are: BC, DE, HL, as seen in the diagram below.



If you now have a quick glance at the list of machine code commands and their hex codes at the back of the book, you will notice under the heading 'MNEMONICS' that there are a large number of commands which deal directly with these registers. You will come to realise that fully understanding the registers, and their function, will be of more use to you than anything else in machine code programming. Another facility of the Z80 CPU that again will prove to be invaluable, is its ability to use the registers as single-byte registers, i.e., with a range of Ø to 255 alone, or Ø to 65535 in cominbination. This is at the core of the computer's versatility.

Shortly we shall look at how to load numbers into these registers, and how to add to and subtract from them. Later we shall also be looking at the ways in which more complicated manipulations of these and other registers can be achieved. Whilst other registers are of less use to us, and some are essentially reserved for the computer's own purpose, there will be times when accessing these registers from our programs will be of considerable advantage.

Probably the simplest way in which to grapple with the concept of 'registers' is to think of the computer as a person who is wearing a coat. This coat has seven pockets, each of which has its own discrete label. The computer can quite easily take items out of these pockets, and put other items into them. To do this it is necessary to 'command' the computer because it has no capacity for thinking on its own, and so it must be instructed most precisely. The computer can only understand commands which are given to it in the form of numbers, or codes. The language, in which commands are solely in numbers, is known as MACHINE LANGUAGE. This is, in itself, the basis of machine code. Unfortunately, whilst it is reasonably easy to put commands into the memory of the computer in the form of numbers — using the computers 'POKE' facility — trying to remember the code for a particular command is not particularly easy for mere mortals. Do not despair just yet though, because, this having long been a problem for others also, a solution is readily available. The answer to our problem comes in the form of 'ASSEMBLY LANGUAGE'. Assembly Language consists of a set of instructions which represent exactly what the codes in machine language mean, but instead they are more easily understandable to us. For example, an instruction in machine language may look like this, Ø11Ø1Ø11, whilst the assembly language equivalent might be, LD A, 1.

Whilst this may not seem to be of major significance just at this point, what ought to be immediately apparent is that our normal facility for making sense of 'groupings' of letters and digits, which we use all the time, allows us to cope rather more easily with the Assembly Language version than the string of binary digits. This certainly becomes more apparent when trying to commit any number of these to memory (yours, not the computer's!). These 'humanised' representations of binary commands are conveniently referred to as 'MNEMONICS'.

Unfortunately, we are not able to type the mnemonics straight into the computer — simply because the computer itself does not understand our mnemonics — instead we have to convert them into a form that can be understood by the poor old computer. This means, one way or another, converting back into binary. There are two basic methods of doing this. The time-consuming, but perfectly effective way of doing this is simply to use the conversion tables at the end of the book. An

easier, and therefore speedier method, is to use a ready-made ASSEMBLER program.

As we proceed through the book I will of course be introducing each of the relevant commands, together with its mnemonic. I will be explaining its meaning and its use, and when working through the given examples you will be effecting the conversions from 'mnemonic' to 'code', and then loading into the computer using the 'loader' provided. You would doubtless find an Assembler of considerable assistance, though you do not necessarily have to dash right out and buy one. Should you decide that you want to cut out some of the labour, and in any event have an Assembler available when I deal with them in more detail later in the book, then you can look up the supplier of the Assembler I will be using at the back of the book. It might not be a bad idea at least to get your order placed.

It is important to remember that assembly language is not an adaptation of machine code — as is Basic. In assembly language there is only one mnemonic for each machine language instruction, and, vice versa, there is only one machine code instruction for each mnemonic in assembly language. We can therefore say that assembly language is, in effect, equivalent to machine language. Nearly all the mnemonics are abbreviations for the operations which a particular machine language command controls. It is, therefore, nearly always a simple matter to convert this abbreviation back into an instruction. For example: INC HL is an abbreviation for increment HL. Similarly, LD A,Ø is an abbreviation for, 'load A with Ø'. The equivalent machine language command for these instructions are 23 and 62,Ø respectively.

Different people elect to list their programs in different ways, i.e. in machine language or in assembly language. It is much easier to understand a program if it is in assembly language, but an assembler for your Spectrum is reasonably expensive in the region of £5 to £9. However if you compare this expenditure to the amount of time and energy which you will save typing in, plus the amount which you will learn by actually being able to understand what is written, the assembler is indeed a very useful aid. If you elect not to use an assembler, or do not have one for any reason, then it is still easy enough to load in programs into your machine via a 'hex loader'. To make this possible, with all the programs which have been used in this book I have listed them in assembly language and machine language, so that whichever way you decide to enter the program, it is still possible for you to look at the assembly language version and work out what it is doing.

Remember that, because machine code is based around the CPU itself, which in this case is the Z80, if you can write machine code for the Spectrum, it will not take you much effort to be able to write machine code on the ZX81, the ZX80 or the Lynx — or indeed any other computer which uses the Z80 CPU. This will be particularly important to bear in mind when deciding which computer to buy if you progress from the Spectrum — or if you decide to add to your stable of micros! For example, should you decide to buy a BBC, or a Commodore VIC20 machine, then you would have to learn again quite a lot of the machine code so as to be able to use it on those computers, but on the ZX81 or the Lynx there would be little to understand before being able to write equally good machine code programs on these computers. This is not the only consideration when deciding whether or not to get involved with a new computer, but could be a factor usefully borne in mind.

OK, now for a break. Here is an interesting program that acts as a simple example of what machine code actually is. Please take note of the two sections of this program — the assembly language listing and the machine language listing. This program scrolls the pixels left.

```
org 30000
30000 21 FF 57      ld hl,22527
```

```
30003 E5            push hl
30004 11 20 5B      ld de,23328
30007 01 20 00      ld bc,32
30010 ED B8         lddr
30012 E1            pop hl
HL=BOT. ADDRESS OF BOT.BLOCK
B=NO.of blocks
30013 06 03         ld b,3
A
30015 C5            push bc
30016 06 08         ld b,8
B
30018 C5            push bc
30019 E5            push hl
30020 06 08         ld b,8
C
30022 C5            push bc
30023 E5            push hl


30024 E5            push hl
30025 D1            pop de
30026 25            dec h
30027 01 20 00      ld bc,32
30030 ED B8         lddr
30032 E1            pop hl
30033 25            dec h
30034 C1            pop bc
30035 10 F1         djnz,C
30037 24            inc h
30038 11 E0 06      ld de,1760
30041 E5            push hl
30042 19            add hl,de
30043 D1            pop de
30044 01 20 00      ld bc,32
30047 ED B8         lddr
30049 E1            pop hl
30050 11 20 00      ld de,32
30053 ED 52         sbc hl,de
30055 C1            pop bc


30056 10 D8         djnz,B
30058 11 00 07      ld de,1792
30061 ED 52         sbc hl,de
30063 E5            push hl
30064 11 20 00      ld de,32
30067 19            add hl,de
30068 E5            push hl
30069 D1            pop de
30070 E1            pop hl
30071 E5            push hl
30072 01 20 00      ld bc,32
30075 ED B8         lddr
30077 E1            pop hl
30078 C1            pop bc
```

```
30079 10 BE         djnz,A
30081 11 20 00      ld de,32
30084 19            add hl,de
30085 EB            ex de,hl
30086 21 20 5B      ld hl,23328
30089 06 20         ld b,32

D
30091 7E            ld a,(hl)
30092 00            nop
ld a,0 for no wrap
30093 12            ld (de),a
30094 1B            dec de
30095 2B            dec hl
30096 10 F9         djnz,D
30098 C9            ret
```

SIMPLE COMMANDS ON THE REGISTERS

You are already familiar with the ways in which the single-byte registers can combine into Register-Pairs. Now is probably a good time to remind you that single registers can cope with numbers between Ø and 255, whilst register pairs can handle numbers in the range Ø to 65535. Let's make a start by looking at how to load numbers into single-byte registers.

Consider, for example, that you want to put a number — say 11 — into one of the registers (or 'pockets'). To do this you must load that number into the 'pocket'. This is done by a simple command known as LD, which simply is an abbreviation for 'LoaD'. Given that there are a number of different 'pockets' or registers, there has to be an equal number of machine language codes for the instructions, i.e. one for each. The codes for the particular loading mnemonics are shown below:

OP CODE	HEX	DECIMAL
LD A,xx	3Exx	62,xx
LD B,xx	Ø6xx	6,xx
LD C,xx	ØExx	14,xx
LD D,xx	16xx	22,xx
LD E,xx	1Exx	3Ø,xx
LD H,xx	26xx	38,xx
LD L,xx	2Exx	46,xx

The first part of the process is to identify which register is to be LoaDed. The second part is to specify the number to be loaded to the register chosen. This means identifying A, B, C (or other register) and specifying the number to be inserted. The Assembly Language representation of wishing to LoaD the A register with the value 11 would look like this: LD A,ØB. However, before this could be executed an assembling program would have to be used in order that the computer could understand what it was being asked to do. Now, if a hex loader was being used — which I recommend that you do for the time being so that you will find it easier to understand just what is going on — then it would be necessary to enter first the number 3E. This tells the computer that a number is to be loaded into Register A. Follow this with the number ØB and it will understand that this number is to be loaded to the A Register.

Note: It is essential to remember that whenever a code for command or an assembly language command is given and there are either two X's, four X's or the letters 'dis' after the command, this means that either one byte or two bytes of your own choice can be put here. For example, when doing the command LD A,XX, the two X's are to signify that we should put a single byte number after the command LD A, to load into the register A.

We could equally as easily load a number into any of the registers A, B, C, D, E... using this method, but also remember the need to use a different code or assembly language mnemonic for each different register (see above). Before going further it is worth our while actually seeing how all this works. There are now two important points which must be borne in mind (though it is not crucial to understand them at this point), before we can go on to create and use a machine code program. The first is that at the end of a machine code program it is necessary to put the code for the command RETurn. RETurn works exactly in the same way when you are doing a GOSUB routine in Basic. For example, if you GOSUB

1000 in a BASIC program, and come to the end of the subroutine at 1000, you will put a command RETURN at the end of the routine to indicate to the computer that you wish to go back to the statement just after the GOSUB 1000 instruction. So, if we go into a machine code routine, from BASIC, by using the BASIC command USR, which stands for USeR SubRoutine, then it is necessary for us to include a RET instruction at the end of the machine code routine, so that it will then return to the program and go to the next available command in BASIC. We shall shortly be looking at why this is so, but for the time being it is only important that you realise this. The machine language code for RETurn is C9. The second point to remember is that if you go into a machine language program by the command PRINT USR xxxxxx, then the number which is printed when the user subroutine is finished is the contents of register pair BC. This fact we can use now to great advantage.

The program below, simply loads the registers B and C with 0 and then RETurns to BASIC, printing 0 when it returns. To enter this program, use the machine code loader program, or hex loader program already provided, and simply type in the hex numbers given. When the program has been entered into the memory of the computer, do not attempt to RUN this program. Simply type STOP and enter the command PRINT USR 30000 directly from BASIC. In this way the machine code routine will return back to command mode, in BASIC, but printing the contents of BC at the same time. Once you have done this try altering the program by putting different numbers into registers B and C, and working out how the answer will be displayed. Remember, that you are using a two byte number, with a maximum of 255 decimal in each byte.

OP CODE	HEX
LD B,00	0600
LD C,00	0E00
RET	C9

The natural progression is to proceed from just loading

numbers into registers to the similar business of loading the contents of one register into another. We have to have a different machine language code for each of these operations (e.g. to load register B into register A), resulting in a large number of different codes, so we shall, at this point, concentrate on loading different registers into register A for ease of explanation and understanding. The principle for all the other registers is exactly the same, except that the machine language codes and the mnemonics are different. Therefore, understanding one is understanding all.

Should you wish to load the contents of register B into register A then this can be accomplished by the simple command LD A,B (meaning LoaD A with B). If we now go back to our earlier analogy of the computer as a person wearing a coat, we can think of this operation as:

1. Looking in pocket A and taking out the contents and throwing them away.

2. Looking in pocket B, finding out how many articles there are in pocket B, but not removing them.

3. Picking up the same number of articles as there were in B and putting them in pocket A.

Now there are the same number of articles in pocket A as in pocket B. It is important to note that the initial value of the contents of A has no bearing on the final outcome of the operation.

You can load the contents of any of the other registers into register A. The mnemonics and the hexadecimal machine language codes for these instructions are shown below:

OP CODE	HEX	DECIMAL
LD A,A	7F	127
LD A,B	78	120
LD A,C	79	121
LD A,D	7A	122

LD A,E	7B	123
LD A,H	7C	124
LD A,L	7D	125

The next thing to do would be to look up the mnemonics and the machine language codes for instructions which load other registers into the BC registers, and use these so that the answer will be displayed on the screen on returning to Basic.

A Cash Register Still Won't Help!

Chapter 5
DOING YOUR SUMS ON THE REGISTERS

Now that we know what the registers are and how we get numbers into those registers, whether by loading them in directly or by loading one register to another, it is necessary now to look at how we can manipulate these registers. If you have ever done any BASIC programming then you will know how pointless it is simply to be able to assign a number to a variable but not to be able to change it arithmetically. That is to say, that to 'work' on a number at all it is necessary to be able to add to or subtract from it. This is what we shall look at in this chapter. We shall also see what happens when a number is either too large or too small for the register to handle. This will lead us on to explore the 'CARRY FLAG' and associated items of information held within the computer.

By now you will have probably started to think about how we are going to be able to add two registers together, and whether or not we will be able to do this in precisely the same way as we do in Basic, where to add the contents of variable XY to variable AB one simply writes: LET AB = AB + XY. The answer would then be held in AB. Fortunately it is possible to perform a simpler operation in machine code, except that the instruction itself is totally different. If we want to add the contents of register pair DE to the contents of register pair HL, instead of writing LET HL = HL + DE (which you would do in Basic), it is necessary only to use ADD HL, DE. The example below shows exactly what would happen if we put this instruction into a program. It is worth noting that the first two lines hold the two numbers to be added together; the third line of that instruction actually does the operation, and lines 4 and 5 load the result into register BC. This has to be done using two separate commands — adding the two single registers together — because there is no such instruction as LD BC,HL, and the last command, which is RETurn, simply takes the computer back to BASIC. I recommend that you try this program right away:

```
ORG 3ØØØØ
3ØØØØ  21  Ø1  ØØ   LD HL,ØØØ1
3ØØØ3  11  ØØ  8Ø   LD DE,32676
3ØØØ6  19           ADD HL,DE
3ØØØ7  44           LD B,H
3ØØØ8  4D           LD C,L
3ØØØ9  C9           RET
```

To our advantage is the fact that the maximum number that a register pair can hold is over 65ØØØ, so it is usually fairly safe to presume that two numbers added together will not go over this limit. However, it is useful to know what would happen if this were to occur. The short program below should make this clear. Try it and find out the answer.

```
org 3ØØØ
3ØØØØ 21 Ø1 ØØ      ld hl,ØØØ1
3ØØØ3 11 FF FF      ld de,65535
3ØØØ6 19            add hl,de
3ØØØ7 44            ld b,h
3ØØØ8 4D            ld c,l
3ØØØ9 C9            ret
```

Before telling you the answer, let us just quickly go through the example. The first command loads the register pair HL with 1. 'Yes', I hear you cry, but the number after 21 in the OP CODE is not 1, it is Ø1 ØØ. Now here we come up against an awkward point which you must get to grips with before going any further. When we load in a number to a register pair, instead of putting the high byte before the lower byte, we do the reverse. Hence, to load the register HL with 1, it is necessary to type Ø1ØØ. The high byte part (the nought nought) goes in after the low byte (or one). If you are not quite sure what is meant by

'the high byte' and the 'low byte', it may be worth just going back to the section earlier on in this book where this was first explained. Just to reinforce your views, here are a few quick examples of short routines which use this method. Fortunately loading the high byte in after the low byte in a machine code program is not uncommon, and so it ought soon to 'come naturally', Now try out these short routines.

1.
```
org 3ØØØØ
3ØØØØ Ø1 ØØ Ø1      ld bc,Ø1ØØ
3ØØØ3 C9            ret
```

The answer that we want is 1, but in this first example we have loaded it in to register pair BC with the high byte and the low byte the 'wrong way round'. For this reason the answer, instead, comes out to be 256. When writing in assembly language it is customary to write the number in hexadecimal, and in the right order, i.e. with the high byte before the low byte. As you can see in this first example, when it came to converting this into hexadecimal code, the high byte was again put before the low byte. WRONG in this instance.

2.
```
org 3ØØØØ
3ØØØØ Ø1 Ø1 ØØ      ld bc,ØØØ1
3ØØØ3 C9            ret
```

This time the high byte has been put after the low byte in the hexadecimal code listing, hence the answer is correct — it is 1.

3.
```
org 3ØØØØ
3ØØØØ Ø1 FA 7B      ld bc,64123
3ØØØ3 C9            ret
```

Again, in this example we see that the high byte and the low byte have been put the wrong way round. Therefore the answer is totally incorrect. Now, before you actually run the next routine, number 4, try and work out, in decimal, what the result should be.

4.

```
org 30000
30000 01 7B FA       ld bc,64123
30003 C9             ret
```

Now quickly, just to recap before going back to the example we were doing earlier; we can say that, when loading a number into a register pair, then the high byte goes after the low byte in the hexadecimal machine language coding. This is another example of how an assembler could help us, in that it is possible simply to type in the number in its high low byte configuration.

Now, back to the example we were doing earlier. We have already established that the first command loads the register pair HL with 1. The next command loads the register pair DE with FFFF in hex, which as you might remember, is equivalent to 65535 in decimal (which in turn is the highest number that can be used in a register pair). If we now add the highest number we can use to 1 then we will get an 'overflow condition'. In BASIC, if an overflow occurs, an error message is shown and the program stops. In machine code this does not occur, because there are no error codes, it simply goes wrong! In this particular case all that happens is that instead of it going to 65536, which is out of range, it simply goes back to zero. Also, if an overflow does occur, then the computer remembers this fact by setting the 'carry flag' to 1. Do not worry about what the carry flag actually is at the moment, we shall be coming to that very shortly.

There are two important points to remember about adding together the two register pairs. Firstly, you can only add another register pair to HL. Also only register pairs may be added to register pairs. A single byte register may not be added to a register pair, and vice-versa. Below is a list of the hexadecimal codes for the operations of adding a register pair to a register pair. You may wonder why it is necessary to have a command to add HL to itself, but this is in fact a very simple way of doubling the value of HL. It is also worth noting that

each of these commands takes only one single byte. Compare this to the BASIC equivalent and you will quickly see the difference as the BASIC equivalent takes over ten times as much memory!!!

```
org 30000
30000 09                ld hl,bc
30001 19                ld hl,de
30002 29                ld hl,hl
```

Now let us look at adding single registers together. Remembering that single registers can only hold numbers in the range Ø to 255, the possibility of 'overflowing' is much more likely than with register-pairs. What happens when they overflow? Well, they simply ZERO, as do register pairs. The question is, can we do anything about this? In fact we can! Whenever we add two numbers together there either is or is not an 'overflow' or carry). The computer itself sets aside a very special register to handle such things as overflows. This is known as the F register. It is unusual for us to be using the F register, simply because the computer itself monopolises it in order to handle various small items of information. The way in which this is achieved, is by separating each single bit off, then allowing for them to be used separately. In your reading you will come across reference to the 'flag' having been 'set' or 'reset'. This is referring to one single bit which might be 'set' — meaning 'turned to 1', or 'reset' — meaning 'turned to Ø'. This then is the CARRY FLAG. The extremely useful thing about the carry flag is that whenever two single byte registers are added together, and the answer is greater than 255, then the carry flag will be set (to 1). If of course the answer is not greater than 255 then the carry flag is reset (set to Ø). We cannot access the F register directly hence it is not possible to change the flags directly, though it is relatively simple to access the contents of these bits.

It is possible to add any other register to register A, including itself, (though you cannot add any register to a register other than A). You will find throughout your learning about machine code that the A register, which is often called the

Accumulator, is featured very extensively. There is no particular reason for this, except that it is the first letter of the alphabet — and the first letter of Accumulator — and so is an easy one to remember. Many of the important commands use this register, just as, with register pairs, the HL register is mostly used. As well as adding another register to register A it is also possible to add a number to register A, but remember that this number must be in the range of nought to 255. Below area a list of all the different ADD commands and their equivalent machine language codes. These numbers are in succession, hence it ought to be reasonably easy to remember them.

OP CODE	HEX	DECIMAL
ADD A,(HL)	86	134
ADD A,A	87	135
ADD A,B	80	128
ADD A,C	81	129
ADD A,D	82	130
ADD A,E	83	131
ADD A,H	84	132
ADD A,L	85	133
ADD A,xx	C6xx	198,XX

Adding single-byte registers together is very similar to adding two register-pairs together, as the examples show:

Example 1:

```
org 3ØØØØ
3ØØØØ 3E Ø5              ld a,5
3ØØØ2 Ø6 Ø6              ld b,6
3ØØØ4 8Ø                 add a.b
3ØØØ5 4F                 ld c,a
3ØØØ6 Ø6 ØØ              ld b,Ø
3ØØØ8 C9                 ret
```

Example 2:

```
org 3ØØØØ
3ØØØØ 3E ØE              ld a,14
3ØØØ2 C6 Ø6              add a,Ø6
3ØØØ4 4F                 ld c,a
3ØØØ5 Ø6 ØØ              ld b,Ø
3ØØØ7 C9                 ret
```

In example 1 the first two instructions simply load 5 and 6 into registers A and B respectively. Command 3 adds the contents of register B into the contents of register A and the result remains in register A. The next two lines simply make it possible for us to display the answer using the simple PRINT USR command in Basic.

In the second example we are simply adding 6 to the contents of the accumulator or register A before we load the contents of A into register C and load B with Ø and RETurn so that the number displayed after the command PRINT USR will be the contents of BC — or in other words, the answer to A + 6. An ADD instruction will always reassign the carry flag. As mentioned before, if there is no 'carry' the FLAG will be set to Ø. If there is a 'carry' it will be set to 1.

So far we have not been able to use this result in any way. The simplest way in which to use it is via the ADC command. This command means "ADD with CARRY" and this is how it works. Suppose the machine comes across an instruction ADC A,B. It will take the contents of register B, add the contents of register A, and leave the answer in register A as in the previous instruction ADD A,B. It will also add the carry flag to this new number. Because the carry flag has not yet been set for this instruction, the number which is added to the answer is the status of the flag previously. Having done this it will store the result in register A, reassigning the carry flag. Hence, if the carry flag has been set by an addition earlier on in the program, and has not been changed since then, the result of the ADC command will be affected by this. At first sight this might appear to be rather more of a hindrance than a help to your programming. However, if you cast your mind back to early school-days when you first learned how to 'add up', it will quickly become evident that this is both a logical and most useful facility. It simply represents the 'carry one' principle in simple addition. What we remember is that first we add up the right hand column and if there was an overflow then a 1 must be added to the next column to the left. For example if we were adding up the numbers 14 and 7, then first we would add up 4 and 7, the result will be 11, so put one underneath the right hand column and 'carry one'. We then add up the left hand column which equals one and zero and add on to that any number which has been carried, in this example 1, hence the answer is 2 and this is put in the left hand column. We can now read the result from left to right reading 21 or twenty one.

The example below shows how we can use this function in a machine code program. Because the program is reasonably long, and quite complicated, do not worry about the explanation until you have typed it in and seen that it works — I shall explain it in the next paragraph. For now just make sure that what you type in is precisely that which you see on the page.

```
org 30000
30000 16 33          ld d,51
30002 1E 85          ld e,133
30004 26 7B          ld h,123
30006 2E C7          ld l,199
30008 7D             ld a,l
30009 83             add a,e
30010 6F             ld l,a
30011 7C             ld a,h
30012 8A             adc a,d
30013 67             ld h,a
30014 44             ld b,h
30015 4D             ld c,l
30016 C9             ret
```

Command 1 assigns the value of 51 to register D. Command 2 assigns the value of 133 to register E. Command 3 assigns the value of 123 to register H and Command 4 assigns L with 199. The objective of this program is to add the contents of register pair DE to register pair HL. This is done not by using a single command but by adding the two halves of each pair together. In other words all we want to do is first add E to L and then add register D to register H. Now the first problem which we encounter is the fact that we can only add one register to register A, hence it is necessary to substitute the register L for register A. This is quite easily done by command 5, which is to load A with the contents of register L. When this has been

done we can add E to A. Hence the value of A now equals the contents of register L plus the contents of register E, but we want to get this answer not in A, but in L. We simply overcome this problem by loading the value of A back into L.

Because we have already added the two low bytes together, by instruction 6, the carry would have already been set if there was an overflow — which there was. We now go through almost the same procedure to add the high bytes together i.e. to add byte D to byte H. This time however we use the command ADC — and add with carry. What happens is that we add D to A, as we would have done earlier, but by this time we have also added the contents of the carry flag. So, if when we added L and E together the result was greater than 255 we would now be adding an extra 1 to the value of the high bytes. This way we can obtain an accurate result. Commands 11 and 12 are simply our favourite little commands which allow us to load the contents of HL into BC so that it can be displayed on the screen after the PRINT USR in BASIC. Another point to notice about this program is that the two commands which are after the ADD A,E but before the ADC A,D command do not actually affect the contents of the carry flag. If you are going to use this type of instruction it is useful to know which commands alter the contents of the carry flag, so that you know what result to expect. For this reason, at the back of the book, you will find a list of all the commands showing whether or not they affect the carry flag.

By now you ought to be able to understand the difference between ADD and ADC. Here are the codes and their mnemonics for the different combinations.

OP CODE	HEX	DECIMAL
ADC A,(HL)	8E	142
ADC A,A	8F	143
ADC A,B	88	136
ADC A,C	89	137
ADC A,D	8A	138

ADC A,E	8B	139
ADC A,H	8C	140
ADC A,L	8D	141
ADC A,xx	CExx	206,XX
ADC HL,BC	ED4A	237,74
ADC HL,DE	ED5A	237,90
ADC HL,HL	ED6A	237,106

It is not directly possible to add a constant to a register pair, but it can easily be accomplished by loading a register pair with the number which you wish to add to HL and then adding the other register pair to HL. The result in HL would equal HL plus the number. For example:

```
org 30000
30000 11 39 00       ld de,57
30003 19             add hl,de
```

This method has the disadvantage that it requires the use of the register pair DE, which you may want to use for other operations. Another way of achieving the same objective is shown below, but this time the only register which is altered, apart from HL, is register A. Take note that in this example I have again used the 'add with carry' command in the same way as we used it in the first example — so that if the low byte overflows it will carry into the high byte, hence making the result accurate.

```
org 30000
30000 7D             ld a,l
30001 C6 39          add a,57
30003 6F             ld l,a
30004 7C             ld a,h
30005 CE 00          adc a,0
30007 57             ld h,a
```

I imagine that you might well have done enough machine code instruction work for now, so here is a machine code program to type in and use. Although it has some of the commands that we have already looked at, it is not essential that you should be able to understand the program at this stage — so do not be upset or worried if there still remains something of a mystery —

though by the end of the book you may wish to look back and discover how it actually works. The idea of putting in this program is for you to be able to use it, see the effect of this piece of machine code, and have a break from learning.

```
org 30000
30000 21 00 40       ld hl,16384
30003 06 C0          ld b,192
CHANGE B & HL FOR DIFFERENT
BLOCKS OF SCREEN
A
30005 0E 20          ld c,32
B
30007 5E             ld e,(hl)
30008 CB 1E          rr (hl)
30010 23             inc hl
30011 0D             dec c
30012 20 F9          jr nz,B
30014 CB 43          bit 0,e
30016 28 09          jr z,C
30018 E5             push hl
30019 11 1F 00       ld de,31
30022 ED 52          sbc hl,de
30024 CB FE          set 7,(hl)
RES 7,(HL)FOR NO WRAP AROUND
30026 E1             pop hl
C
30027 A7             and a
30028 10 E7          djnz,A
30030 C9             ret
```

As a prelude to the next section in the book, look through this listing and notice all the commands which have brackets around some part of them.

LOADING THE CONTENTS OF AN ADDRESS LOCATION INTO A REGISTER

Earlier, as you may remember, we looked at how numbers can be stored in boxes — or locations in the memory. As you will recall, each of these boxes had a number which constituted the specific and discrete address of the box. The reason for having this address is so that we can access the contents of this box relatively easily. I pointed out how useful it would be to be able to manipulate the number in the boxes, but so far we have yet to talk about how actually to get numbers out of the boxes, or location in memory, and into a register. Obviously, when it is in the register we will be able to manipulate the number properly. Then it should also be possible to load the contents of the accumulator or any other register into a location or address in memory. In this way it becomes possible to save a particular number for future use. Also, using this method, it is possible to have an enormously large quantity of numbers all being accessible at once.

Let us consider what we actually want to do. Our task is to load into say the accumulator, or register A, the contents of a particular location. This location will have an address. If we simply wrote Load A with the address, then the computer would mistake the address for a direct number. For example, if we typed LD A,12, then it would be reasonable to expect that the computer would consider that we wanted to load the accumulator with the value 12. But what we want is to load the accumulator with the contents of a particular location. In fact it is quite simple to do this as all we have to do is instead of typing the number after the instruction, we place the number in brackets after the 'comma'. For example: LD A,(2465). The computer will now load the accumulator with the value at location 2465. Because each location can only hold a number between Ø and 255 there is no problem in loading a single byte register with the contents of a location. If we said that at location 2465 there was the number 64, then after executing this command the accumulator would also contain 64. As with all these commands the content of the address is still maintained. In other words, the contents of address 2465 is still 64. This is a very useful command, in that we are now able to take the contents of any location in memory, put it in a register, and manipulate it. For example:

```
org 30000
30000 3A 30 75        ld a,(30000)
30003 C6 29           add a,41
```

The only problem now is how to put the manipulated number back into that address, or into another address. This can quite easily be effected by using a variation of the above command. If you think about what we actually want to do at this stage,

you will realise that instead of loading the accumulator with the contents of our adress, we actually want to load the address with the contents of the accumulator. What we do is to use the command in reverse: LD (ADDR),A taking that the address is ADDR.

Below is a simple example of how this concept works.

```
org 30000
30000 3A 30 75        ld a,(30000)
30003 C6 0F           add a,15
30005 32 30 75        ld (30000),a
30008 C9              ret
```

Going back to the first concept which we used, i.e. that of taking a number from a location, manipulating it and then putting it back in the same location, we can change the colour of the screen for instance – by getting the colour which is there already and manipulating this so that it is a different colour. Because the screen, as far as the colour is concerned, is 704 bytes long, it is necessary to carry this out 704 times. Fortunately it is possible to bring into use a different instruction in order not to have to write out the program 704 times! This we shall come onto a little later on. Now before we go on to the example of a change colour program below, here is a quick task for you: Write a routine which takes the character code of any character out of memory, changes it to the inverse of that character and then puts it back. Here is a quick tip, the inverse of each character is 128 larger than the normal (in decimal). In other words, if you took the code for the character 'a' and added 128 to it, you would now have the code for 'inverse a'.

```
org 30000
30000 21 00 58        ld hl,22528
30003 01 C0 02        ld bc,704
30006 1E 06           ld e,6
loop
30008 7E              ld a,(hl)
30009 CB 87           res 0,a
30011 CB 8F           res 1,a
30013 CB 97           res 2,a
30015 83              add a,e
30016 77              ld (hl),a
```

```
30017 23          inc hl
30018 0B          dec bc
30019 79          ld a,c
30020 FE 00       cp 0
30022 20 F0       jr nz,loop
30024 78          ld a,b
30025 FE 00       cp 0
30027 20 E8       jr nz,loop
30029 C9          ret
```

Note: ld e, "new INK colour"

SUBTRACTION

In machine code, the instructions for subtracting numbers and registers from another register are exactly the same as their counterparts for addition. For this reason subtraction is a very simple concept to understand. There are differences of course. With addition you can overflow, but with subtraction you can 'underflow' — in other words, the number from which you are 'taking' is smaller than the number you are attempting to subtract. Try subtracting 11 from 5 there will be an underflow of 6. If an underflow does occur in subtraction then the carry flag is set to one, but if there is no underflow then the carry flag is reset to 0. The next factor to bear in mind is that there is no positive or negative flag. When an overflow occurs in addition the numbers start adding again from zero. For instance, if you add 5 to 255 then instead of going to 256 and then adding 4, the computer will go back to nought and add 4. In subtraction, it will go back to 256 and then subtract.

Subtraction works in the same way as addition, meaning that the instruction SUB A,B (SUB for SUBtraction) will take the value of register B away from the value of register A, and the result will be stored in register A. The carry flag is then set or reset accordingly.

Because the command SUB only applies to single byte registers, and only the accumulator at that, it is common (instead of writing SUB A,B) for the instruction "subtract B from A" to be written as "SUB B". This may be slightly confusing at first, but you will get used to it quite quickly. As

subtraction is not the simplest of subjects when dealing with an unfamiliar format it may be a good idea at this point to look back at the work done on subtraction earlier in the book, simply to reinforce your understanding. The different assembly commands, and their equivalent machine language, codes are shown below:

OP CODE	HEX	DECIMAL
SUB (HL)	96	150
SUB A	97	151
SUB B	90	144
SUB C	91	145
SUB D	92	146
SUB E	93	147
SUB H	94	148
SUB L	95	149
SUB xx	D6xx	214,XX

Below is a quick example of how the subtraction statement can be used:

```
org 30000
30000 3E 0B       ld a,11
30002 06 07       ld b,7
30004 97          sub a,b
30005 4F          ld c,a
30006 06 00       ld b,0
30008 C9          ret
```

It is possible to subtract numerical constants from the A register. For example, the instruction SUB A,100 will subtract 100 from the number stored in register A. The result is then stored in register A. You should note that though there are ADD instructions for register pairs there are no subtraction commands for register pairs. Below is a short example of subtraction in use.

```
ORG 30000
30000  0600     LD B,00
30002  3E58     LD A,88
30004  D633     SUB 51
```

```
30006  4F      LD C,A
30007  C9      RET
```

SuBtract with Carry (SBC) on the other hand, will work for register-pairs; but as with ADD and ADC, only the value of register pair HL may be altered, or the contents of single register A altered. SBC A,C will subtract the value of register C from the value of register A, and will then subtract the value of the carry flag from this result and store it in register A. Again the command "Subtract with Carry" may be used when subtracting two numbers from each other where there might be an overflow.

Thinking back to the way in which we originally learned how to subtract numbers from each other we can see how this works. If we were to have the number 21 and wished to take 19 away from it, this is how we would proceed. First take 9 from 1. This is not exactly possible — and still keeping the number positive — so the answer is to 'borrow' 1 (value 1Ø) from the column to the left. We are now 'taking 9 from 11', giving "2" as the answer. It now has to be taken into account that the next column has been reduced by 1, resulting in what is termed "an underflow". It is necessary therefore to take '1 from 2', the result being 1 — but because we have to take the carry flag from this result the answer is Ø and the final answer is 2.

You will be able to gather from the list below, that using "Subtract with Carry" on a register pair requires two bytes, and a single operation on a single register requires one byte — which makes both of these commands very compact indeed. The assembly language codes and the machine language codes are shown below:

OP CODE	HEX	DECIMAL
SBC A,(HL)	9E	158
SBC A,A	9F	159
SBC A,B	98	152
SBC A,C	99	153
SBC A,D	9A	154
SBC A,E	9B	155
SBC A,H	9C	156
SBC A,L	9D	157
SBC A,xx	DExx	222,XX
SBC HL,BC	ED42	237,66
SBC HL,DE	ED52	237,82
SBC HL,HL	ED62	237,98

OPERATIONS ON THE CARRY FLAG

It is often useful, when using instructions such as Subtract with Carry and Add with Carry to be able to determine the contents of the carry flag. Unfortunately it is not possible to reset, i.e. set the carry flag to nought directly, but it can be done by a simple machine code trick. We know that whenever a number is added to the accumulator, the carry flag is set according to whether or not there is an overflow. If we add Ø to the accumulator then it is not possible that there will be an overflow — THEREFORE BY DOING PRECISELY THIS THE CARRY FLAG CAN BE SET TO ZERO. This is simply because the carry flag is always set to Ø when there is no overflow. This is probably the most useful little command with the carry flag. When using the aforementioned commands it can alter the result quite tremendously if the carry flag is set to 1 and is not meant to be so. Setting the carry flag to 1 is a much easier task, there being an instruction to do this. The instruction is SCF — which means, "Set Carry Flag". The machine language code for this is 37 in hexadecimal.

This concludes the chapter. We have looked at two important ways of manipulating registers, and ways in which we can take numbers from locations in memory and put them into registers. We have also looked at more complicated commands — such as ADd with Carry — and ways in which to overcome problems by setting and resetting the carry flag. I hope by now you will be able to go away and write a short machine code program using these commands.

The Art Is In Getting It In!!

Chapter 6
ASSEMBLERS, DISASSEMBLERS, and DEBUGGING PROGRAMS

In this chapter we shall take a break, from actually learning machine code. Instead we shall be looking at the three main types of program which have been written specifically to help the machine code programmer. Having now gone quite some way towards learning how to write simple machine code programs, you will doubtless soon be ready to embark on rather more adventurous programming. This will lead you into writing longer machine code routines and for this reason it is useful to have some special aids to make life easier. Most of these aids, or utilities, are themselves written in machine code. Clearly therefore, the programmer who wrote it will likely be someone with considerable knowledge of his subject, and not necessarily given to setting out the operating instructions in the simplest of language. In consequence, when it comes to deciphering the instruction manuals for the assembler debugger or dissassembler programs, the beginner could be very much at a disadvantage. For this reason, at the end of this chapter, I have included a short section carefully explaining exactly how to use a particular example of each of these types of program. It is also useful to note that these examples are all commercially available. Many will be obtainable at retail outlets, but none-the-less you will find that I have included mail order addresses for your convenience at the end of this book.

Choosing utility programs is very much a matter of personal preference and experience. For your clarification all the programs in this book were assembled/disassembled with the aid of the ACS Software programs.

ASSEMBLERS

At this point, it is important that we know exactly what we mean by machine and assembly languages, and are therefore comfortable with most of what has been dealt with so far — so here goes for a quick recap. (If you feel confident that you know exactly what these mean and the differences between machine language and assembly language, then you can miss this section out).

"Machine Language" is made up of numbers and numbers alone. The computer itself understands these numbers and can execute them as commands. Unfortunately, having to remember a large quantity of numbers, and then being able to know exactly what they do, is not a task which is particularly easy for us humans. For this reason "Assembly Language" was formulated. Each command in assembly language corresponds directly to a command in machine language, except that in assembly language these instructions are written in mnemonics which are abbreviations for the actual commands. For example: if we wanted to execute the command "Load register A with the contents of register B", this can be accomplished in machine language simply by entering the code 78, and executing it. But to remember the number 78 and that it corresponds with the above instruction is not an easy task. One way of obviating this is to have a list of all the instructions and all the codes written down for you so that you can look up the particular command and its corresponding machine language code.

The first problem with this is that if we had completed explanations of each particular command they would take up too much room. So instead of doing this we have a MNEMONIC. A mnemonic is simply an abbreviation of its

command. Instead of having "Load register A with the contents of register B" we have the simple abbreviation LD A,B. You will already have found that this abbreviation is far easier to remember than the machine language code, and takes up a lot less space than the complete explanation. Now, if we want to use the method of reference, this can quite easily be done by listing the mnemonic next to the machine language code — (this has been done at the back of this book). Should we learn all the mnemonics for the machine language instructions we would have what is known as assembly language. In other words, assembly language is simply made up of the mnemonics of machine language. For this reason it can be seen that assembly language is not an adaptation of machine language such as BASIC, but there is one and only one mnemonic for each particular machine language instruction and vice-versa.

What we have therefore is a language that we can quite easily understand, and can use it on the basis of conveniently formulated listings. However, whilst what we now have is a more convenient language, it still requires the expenditure of some time and effort to "convert". Why not then look to the clever old computer to do all the hard work for us in converting the mnemonic into machine language code? This is what an ASSEMBLER is able to do for us: a program which assembles mnemonics into machine language code.

Another way of thinking of this action, is looking at the way in which a fast food cafe works. When you go in you will order via a menu, and on this menu there are various dishes, i.e. hamburgers, whatever. When you give the order to the attendant then he or she will not write down the actual order, but rather they will write down a number which corresponds to a pre-defined product. Hence, what they are doing is assembling your order into a form which the people back in the kitchens will understand, and use more quickly.

All the common assemblers for the Sinclair Spectrum work on the principle that, first you type in your program in assembly language, and then when it is complete you instruct the program to assemble it into machine language. This machine language code is then placed in a particular part of the memory depending on where you desire it to be. Of course, it also depends on how much memory your computer has available. An assembler program is a fairly complicated type of program and some of the assemblers on the market are only suitable for the 48K Sinclair Spectrum. This influenced my choice of Assembler in writing this book as ACS programs will work on either the 16K or 48K version.

If you look in any Sinclair or personal computer magazine, you will notice various advertisements for assemblers. The price of an assembler is usually a couple of pounds more than a game program. This is for two reasons; firstly they are, for the most part, more complicated than games; secondly they are also less of a "mass market" product and cannot necessarily be counted on to sell in particularly large quantities. Believe me though, they are well worth paying for. They are an incredibly useful utility for the serious machine code programmer.

DISASSEMBLERS

A disassembler does exactly the opposite of an assembler in that it converts the machine code into assembly language code.

There are two main ways in which you can use a disassembler:

1 — simply as an aid to checking that what you have written into the memory of the ocmputer is exactly what you intended it to be.

2 — to look at what other people have written in machine code, and how they have tackled certain problems.

For example, you may wish to disassemble certain parts of the Sinclair ROM in order better to understand what goes on in there and, if possible, to utilise the routines within it. Also, in many magazines, when a machine code program is listed they

will only list the hex code, or the machine code. I would advise you at a later date to load in some other people's programs and disassemble them, and so you get some idea of how other people write programs. There is much to be gained from looking at how others make use of the various instructions.

With the assembler and disassembler used for this book, it is possible, if you have a 48K machine, to load both of them at the same time. Again, when choosing your own utilities, bear in mind the usefulness of this kind of "compatability". It is not possible to load the assembler and disassembler into a 16K machine both at once, simply because there would not be enough memory. However, in such circumstances it is possible to load just one of them at a time. Because it is possible to load and save our own machine code programs independently of programs already in the computer they can be assembled using the assembler, saved, and then loaded back in with the disassembler for disassembling.

DEBUGGING PROGRAMS

The third and final type of program now to be considered, is not one which can be used when you are first writing a machine code program, but rather it is useful to debug, and to make alterations to the program after it has been assembled. When you buy a "debug" program you are not generally buying one single program. A reasonably good machine code debug program may offer some of the following facilities:

1. The option of being able to run your machine code routine one instruction at a time, and being able to display the contents of all the registers after each step. This is an exceptionally useful facility when it is necessary to locate a bug in a program, or when you wish to see exactly how someone else's routine works.

2. Being able to insert a "break point" somewhere within your program, i.e. to make the program run up to that point and then stop. This command can be used in various ways. You may wish not to know the contents of the registers after each

step in the program, especially if it is a long one, but do wish to know the status of all the registers and the flags after a certain operation, or block of operations have been completed. To continue the program running should be an easy matter, simply by pressing a given key.

3. To be able to change the contents of a particular register whilst the program has stopped at a break point, which can usually be done quite easily. This may be useful in two particular cases. Firstly, if you wish to know what the outcome would be if you loaded particular registers with given numbers. Secondly, it may be useful to be able to simulate a certain condition when values are inputted into the program which are too large for the program itself to handle, but you wish to know whether or not your error handling routines are working properly.

4. The capability to execute your program from the monitor. This may seem an obvious one, but whilst you are in a monitor program it is not always easy to go back into BASIC and use the USR command.

5. The facility to convert numbers from hexadecimal to decimal, within a debug program.

6. Three useful commands which, when used together, make the debug program capable of altering your already assembled machine code program. These allow for the inspection of the contents of any particular address, and to alter it if required. Additionally, with many assemblers, if you keep your finger on the enter key, the contents of subsequent addresses are automatically displayed on the screen, but no new value is entered unless of course you type a number before you press Enter. If you find that you wish to insert a command within a machine code program this is often extremely difficult to do after having assembled the program because, of course, it would be necessary to move up or down one half of the program. With many debug programs inserting a new

instruction is easy to do simply by telling the computer exactly where you wish to insert, and then how many bytes you wish to insert so that space can be created, and finally, the code to be inserted. The complimentary instruction to this is of course "delete", which can be found in most debug programs. This works in exactly the opposite way to the "insert" command.

The above is simply a quick guide to some of what you may find in a debug program, though usually you will find various extra commands from those listed above. The way in which debug programs operate varies according to the program you are using.

Often an assembler or disassembler may be merged into a debug program, or vice-versa. For example, often you will find within a debug program that it has a disassemble function. Another example of this practice is the assembler available from Picturesque. As well as simply being able to assemble code, you are also able to edit it using various sophisticated commands. The monitor program, which is basically a debug program and an assembler, can be used at the same time as the assembler/editor program, and so makes for a very high-powered combination. For simplicity's sake, and so as to make things a little bit cheaper, the assembler/disassembler/debug programs, used for this book were separate programs, and available on separate cassettes.

A simple debug program does not necessarily have to be written in machine code. You may decide that you do not wish to buy a propriety program at this stage. The best compromise is to use the monitor program listed on the next few pages. This will allow you to enter the machine code into the computer and edit it satisfactorily, although you will have to enter it in its machine language code. Conversion from Assembly Language Mnemonics can be accomplished by using a reference section at the back of this book.

When you have typed in this basic program remember to save it onto cassette before testing it, and as soon as you are sure that the program is working correctly then make a separate back-up copy on another cassette, just in case one of the cassettes is damaged in any way.

```
   5 CLEAR 31999
  10 PRINT "     ####Spectrum Moni
tor####""""            by James Walsh
"33"
  30 REM Load M/c
 100 INPUT "What start address(d
ec)?";loc
 110 PRINT AT 5,0;
 120 PRINT loc;"....";
 130 LET n=PEEK loc: GO SUB 1400
 140 INPUT z$
 145 PRINT "....";z$
 150 IF z$="end" OR z$="END" THE
N GO TO 800
 160 IF z$="p" OR z$="P" THEN CO
PY : GO TO 120
 170 IF z$="S" OR z$="s" THEN GO
 TO 280
 190 IF z$="v" OR z$="V" THEN GO
 SUB 1600: GO TO 120
 200 IF z$="j" OR z$="J" THEN LE
T loc=loc-1: GO TO 120
 210 IF z$="n" OR z$="N" THEN GO
 TO 100
 220 IF z$="z" OR z$="Z" THEN LE
T z$="00"
 230 LET ans=(CODE z$(1)-48)*16
 240 IF ans>9*16 THEN LET ans=an
s-7*16
 250 LET l=CODE z$(2)-48: IF l>9
 THEN LET l=l-7
 260 LET ans=ans+l
 270 POKE loc,ans
 280 LET loc=loc+1
 290 GO TO 120
 800 STOP
1400 REM **HEX/DEC subroutine**
1410 LET x=INT (n/16)
1420 LET y=((n/16)-INT (n/16))*1
6
1430 IF x>9 THEN LET x=x+7
1440 IF y>9 THEN LET y=y+7
1450 LET x=x+48: LET y=y+48
1460 PRINT CHR$ x;CHR$ y;
1470 RETURN
1600 REM **INPUT HEX/DEC SUB.**
1605 INPUT "Number to be convert
ed";n
1607 LET a=n
1610 LET x=INT (n/16)
```

```
1620 LET y=((n/16)-INT (n/16))*1
6
1630 IF x>9 THEN LET x=x+7
1640 IF y>9 THEN LET y=y+7
1650 LET x=x+48: LET y=y+48
1660 PRINT a;" = ";CHR$ x;CHR$ y
1670 RETURN
```

Now we come to look at how the monitor program above is used. It is shorter than the commercial programs that we have looked at so far, and is certainly the most straight-forward to operate. The first thing to notice is that line 5 is used to specify RAMTOP by way of the CLEAR instruction. It is therefore necessary to reassign this value, depending upon which part of memory to be used for the machine code routine. Now RUN the program. You will be prompted for the start location to be entered — do so. This start address will be shown on the screen, followed by its contents in Hex. A new value may now be entered (in Hex), remembering only to use CAPITALS. Alternatively, one of the following commands can be used:

P — copy the screen to the printer.
S — leave contents value unchanged and jump to next location.
V — convert a decimal number to Hex.
J — jump back one location.
N — re-start from new location.
Z — let contents value be zero.

Once the value has been entered, or commands P, V, or Z executed, the next location and its contents will be displayed. To bring this process to an end, type "end" when asked for a value. Though this program is short, and relatively simple, it will prove invaluable in the absence of an Assembler or commercial debugging program.

USING AN ASSEMBLER

The assembler can be loaded into your computer in the normal way. If you have both 16K and 48K versions do be sure to load the correct copy for your particular machine. It is important that you do not load the wrong copy. The assembler used in the writing of this book is available from ACS SOFTWARE, and called Ultra Violet.

If you decide to use the same assembler you will probably benefit from some additional explanation of how to use it.

To load Ultra Violet simply use LOAD " ". (The 16K version loads in three parts.) To prepare the assembler for use press any key on the keyboard. This will clear the memory in the computer 'below' where the assembler is being kept. The assembler is kept in 'high memory', well out of the way of BASIC programs. It will remain there and can be called from BASIC.

Having done this you will be in normal BASIC command mode. Machine code can now be entered into a program almost as if it were BASIC. The only difference being that there should be a REM statement before the machine code itself. The line containing the machine code command would look like this: First the line number, then a REM statement, then the assembly language mnemonics.

The first thing to do when using this assembler is to decide exactly where you wish to put the machine code after it has been assembled. There are two easy alternatives; first and probably the most useful is to put it somewhere in memory above the Basic program, but below the assembler.

Note: The assembler itself uses, in the 48K version, addresses 60000 upwards, and in the 16K version addresses 27500 upwards. Hence it is not possible to assemble code directly into these locations, although it is possible to get round this problem as we shall see later.

The second alternative is to assemble the machine code into a REM statement at the beginning of your program. It only has to be at the beginning of the program for convenience. The usual address for the first character after the REM statement in the first line is 23760. If we have decided to put machine code

in a REM statement, which probably is not advisable at this moment, and is unnecessarily complicated on Spectrum, then you must put sufficient 'spaces' or 'characters' after the first REM statement to hold the machine code. Remembering that if you wish to hold a 25 byte machine code routine in a REM statement, then this REM statement must contain at least 25 characters *after* the REM statement itself.

This assembler recognises all the standard Z80 machine code mnemonics in lower case, exactly as they appear in the back of the Sinclair ZX Spectrum Basic programming manual, which was supplied with your computer. There are only a couple of exceptions to this rule, which are not particularly important, and we will look at them later. An important point to note, is that all numbers should be entered in decimal, though they will be listed in hexadecimal after having been assembled.

PUTTING YOUR PROGRAM INTO THE ASSEMBLER

The first instruction of any machine code program to be assembled is the GO command. This is not a Z80 machine code mnemonic, it is purely to tell the assembler that this is where the program to be assembled is going to reside. The next line must contain the actual address at which the machine code must be assembled. This is done simply by the command ORG followed by the address. ORG is not a Z80 mnemonic either, it is only to tell the program where to put the assembled code. ORG stands for "ORiGin". If either of the above two commands have been omitted from a program then an error code will ensue if you try to assemble it. Now you are ready to put in the mnemonics for your machine code program. You may put more than one mnemonic on each line, provided they are separated by semicolons, though it is probably easier and clearer if you have just one instruction per line. So now we have put space at the beginning of the program for the machine code (if of course we want to put it in a REM statement). The computer has been told by the GO instruction where the code to be assembled is to be held, and we have also told it, via the ORG command, where in memory we wish to put the assembled code. Now we are ready to enter the assembly language program. Often, during a program, you may wish to put some sort of remark or explanation of what you are doing. This can be done whilst using an assembler, by putting an exclamation mark after the REM and then typing in the comment. The exclamation mark is used to tell the assembler that the codes or the word after it are not for assembling, but are simply there for the purpose of the programmer. Should you accidentally omit the exclamation mark the assembler will try to convert whatever you have written into machine code, and probably cause the assembler to stop and show an error code.

You can now type in the program to be assembled. This should be done in lower case, which is the 'case' it will be in when first turned on, so that the computer can understand what you are doing.

Note: You may use upper or lower case letters within a REMARK statement, though you may only use lower case in an instruction to be assembled.

When you have typed in all the assembly language mnemonics for assembling you must put the word Finish after your REM statement. This tells the assembler that you have come to the end of the routine which you wish it to convert. Finish is not a Z80 mnemonic.

You should now have a full machine code program in assembly language form, which is now ready for assembly into machine language. Before doing this it is useful just to check that there have been no mistakes made in entering the program. Obviously, if there are, it is a simple matter to edit out the particular line and change the error statement, just as you would in BASIC. To tell the assembler program that you wish it to assemble the mnemonics already typed in, type either RANDOMIZE SPACE USR 6ØØØØ, if you have a 48K Spectrum, or, RANDOMIZE SPACE USR 275ØØ, if you have a

16K Spectrum. Now the mnemonics will appear on the screen alongside their machine language codes. If there are any errors, i.e. statements which the assembler cannot understand, then an error code will be shown and you will be asked to rectify this problem. Please note that an assembler will not check the actual program for you. An interesting point to note, is that when the program is assembling, the listing of the machine code program is printed twice. This is because it is a two-pass assembler. This simply means that it goes through the program twice before putting a final version into memory. The reasons for this will become apparent later.

Now we know exactly what we can do in theory, let us see whether we can put it into practice. The first thing to remember is that we wish to put the machine code, not in a REM statement at the beginning of the program, but say at location 3ØØØØ. The program we shall now assemble is shown below, this is a pixel scroll right routine.

```
                        ld hl,22527
                        push hl
                        ld de,23328
                        ld bc,32
                        lddr
                        pop hl
HL=BOT. ADDRESS OF BOT.BLOCK
B=NO.of blocks
                        ld b,3
A
                        push bc
                        ld b,8
B
                        push bc
                        push hl
                        ld b,8
C
                        push bc
                        push hl


                        push hl
                        pop de
                        dec h
                        ld bc,32
                        lddr
                        pop hl
                        dec h
```

```
                         pop bc
                         djnz,C
                         inc h
                         ld de,1760
                         push hl
                         add hl,de
                         pop de
                         ld bc,32
                         lddr
                         pop hl
                         ld de,32
                         sbc hl,de
                         pop bc

                         djnz,B
                         ld de,1792
                         sbc hl,de
                         push hl
                         ld de,32
                         add hl,de
                         push hl
                         pop de
                         pop hl
                         push hl
                         ld bc,32
                         lddr
                         pop hl
                         pop bc
                         djnz,A
                         ld de,32
                         add hl,de
                         ex de,hl
                         ld hl,23328
                         ld b,32

D
                         ld a,(hl)
                         nop
ld a,0 for no wrap
                         ld (de),a
                         dec de
                         dec hl
                         djnz,D
                         ret
```

The above program is shown in assembly language, the job of the assembler being to convert this into machine language codes and the first operation is to put all of this into the assembler. The first thing we have to do is load the assembler

in. This is done by typing LOAD "" making sure that we load the right side (48K or 16K) for the size of the computer, Depending on the size of the machine, the assembler should take less than a minute to load. If you are using the Ultra Violet assembler, then a large opening screen will be shown, but this will disappear as soon as you press a key, leaving the normal power up message on the screen, i.e. (c) 1982 Sinclair Research Ltd. You are now ready to type in the assembly language instructions via a Basic program. You can now treat the operation as though you were writing a simple Basic program. It is very important to remember that all the assembly language instructions, and the instructions used to tell the assembler itself what to do, have to be entered in a line, but after a REM statement. This is simply so that the machine itself cannot execute these instructions as they are only recognisable to the assembler and not to the BASIC.

The assembler must now be told that there is a program in assembly language ready to be assembled. This is done by inserting the instruction "GO" as the first line of the program. For example: 5 REM go. Next we must specify whereabouts in memory we wish to put the assembled machine code routine. This is done by the "ORG" instruction, followed by the address at which we want the first byte of assembled code; the rest of the code will then be put sequentially after this address. So, for example, if our routine is 1Ø bytes long, and we ask the computer to start to assemble at the location 28ØØØ, then it will be put into addresses 28ØØØ to address 28ØØ9, (ten bytes in all). Because we wish to have the above program assembled into locations 3ØØØØ onwards, the next line of BASIC program which we enter is: 8 REM org 3ØØØØ.

We are now, in fact, almost ready to type in the assembly language instructions, but before doing so it is quite useful to insert a comment at the beginning of the program so that, recognition is possible when we come to look at it at a later date. The way in which we enter a comment into a program, is by putting an exclamation mark after the line number and the

REM statement, and then entering the comment. For the above program it would be sufficient to state that it is a scroll-down program and we would do this by: 12 REM ! Pixel Scroll — down.

The stage is now set, as you might say, for the actual assembly language instructions to be entered. Remember, as you would in a BASIC program, to give line numbers throughout the routine, and do not forget to put the REM statement before each instruction. The final command which we must give the assembler itself, is to tell it that we have finished the code to assemble. This is done by use of the command "Finish". This can be put into a line just as we have done above: Line number REM finish. Having entered this into the computer, assembly can now begin.

If for any reason you have had problems typing this in, and in case you wish to check what you have typed, below is what your final Basic program should look like. Do not worry about whether the line numbers are exactly the same as in mine, just ensure that the actual instructions are in the same order.

```
   1 REM go
   2 REM org 3ØØØØ
  1Ø REM ld hl,22527
  2Ø REM push hl
  3Ø REM ld de,23328
  4Ø REM ld bc,32;lddr;pop hl;!H
L=BOT.ADDRESS OF BOT.BLOCK
  5Ø REM !B=NO.of blocks;ld b,3;
A;push bc;ld b,8;B;push bc;push
hl;ld b,8;C;push bc;push hl;push
 hl;pop de;dec h;ld bc,32;lddr;p
op hl;dec h;pop bc;djnz,C;inc h;
ld de,176Ø;push hl;add hl,de;pop
 de;ld bc,32;lddr;pop hl;ld de,3
2;sbc hl,de;pop bc;djnz,B;ld de,
1792;sbc hl,de;push hl;ld de,32;
add hl,de;push hl;pop de;pop hl;
push hl;ld bc,32;lddr;pop hl;pop
 bc;djnz,A
  6Ø REM ld de,32;add hl,de;ex d
e,hl;ld hl,23328;ld b,32;D;ld a,
(hl);nop;!ld a,Ø for no wrap;ld
(de),a;dec de;dec hl;djnz,D;ret
1ØØØ REM finish
```

Assembly is achieved by accessing a machine code routine which has already been loaded into the computer. As there are two versions of this program, there are two different addresses at which this assembling routine is kept. If you have a 16K Spectrum then you must type the following command:

RANDOMIZE USR 275ØØ.

If you have a 48K Spectrum then you must execute the command:

RANDOMIZE USR 6ØØØØ.

Important: Please remember that the above instructions for assembly, and also the way in which we enter our assembly language routine into the assembler, is only applicable if you are using the ACS Assembler. There are various assemblers on the market at the moment, and there are likely to be even more by the time that this book is available and it is impossible for me to outline the ways in which they all work as there are often vast differences in their operation. I have elected to concentrate on the use of one particular assembler as I believe that this is one of the easiest to use. If you already have an assembler, or you decide to purchase one other than the ACS that I am using, then it will be necessary for you to carefully work through its instruction manual. This also means that the last listing shown is probably not usable with another assembler. Of course the routine which was listed a little earlier will run exactly the same regardless of whose assembler you use.

Now back to the task in hand. We have the BASIC program ready, and we know how to instruct the assembler program to assemble the instructions which we have typed in — so how about getting on with it? Having executed the command for assembly, shown above, a multi coloured display will be shown on your screen (it will be only multi-grey if you are using a black and white television!). The assembly is not yet complete. If you press key "P", then a listing of your machine code routine will be put onto the printer. Press the space key should you wish to abort the assembly – pressing any other key will cause the assembly to continue. It is essential to remember that you can only use the "P" command for printing at the printer if the machine code routine is being displayed for the second time, or is in its second "pass". Having pressed the ENTER key down, and a "No Error" message shows at the bottom of the screen, you are ready to execute the machine code routine from address 30000 onwards in the normal fashion (RANDOMIZE USR 30000 or PRINT USR 30000 etc.).

An error in your program will occasion the following to occur:

If Go, Finish or Org are faulty, then an error will be given before assembly starts.

If an error is detected during the assembly, then the assembler will stop with one of two error messages. The first is a flashing error message which shows you the line number, statement number and type of instruction where the error was detected. This makes it nice and easy for you to go back in your BASIC program, and so find where the error occurred. The second error message is one of the Sinclair messages. There are three possibilities. If you enter a wrong number, i.e. you try to store a number greater than 255 in a single-byte register, or greater than 65535 in a register pair, the assembler will repeatedly subtract either 256 or 65536 from it until it gets a sensible number that can be used. Unfortunately, this will not happen if you have made the displacement for a relative jump too great. Whilst we have not yet dealt with relative jumps, though we shall be doing so shortly, this is an important fact to remember later on. In the event of it not being able to find a sensible result, i.e. with a relative jump, then error "B Integer out of Range" will be given.

There are also other cases in which errors will be detected, but these are not important at the moment.

Now that we have a machine code routine in memory, which should be saved so that if any fault occurs during execution we

can simply reload the program. Saving of a machine code routine is well documented in the Spectrum manual itself, but I shall quickly go over it.

All that is necessary is to type SAVE "Name" CODE 30000,100. This simply tells the computer to SAVE the machine code program (it knows that it is machine code because you typed the instruction CODE followed by two extra numbers), at addresses 30000 on to addresses 30000 + 100, under the name inserted between the quotes. For example, if we wish to save the above machine code routine, under the name pixel-d, then we would type the following: SAVE "pixel-d" CODE 30000,100. Do not forget that we are only saving the machine code program, and not the assembler or the BASIC program used for entering the assembly language instructions into the assembler program itself. Loading back in again is even simpler than saving, just type: LOAD "pixel-d" CODE. This will load the machine code program which has the name "pixel-d" into the addresses from which it was saved. When the loading is complete a "0" error message will show at the bottom of the screen.

DISASSEMBLERS

Often it is useful to be able to do the exact opposite to what we have just done. In other words, instead of assembling a list of assembly language mnemonics into machine language code, we may want to convert the machine language code to assembly language mnemonics. This is often true when we have a routine in memory which we have not written ourselves, or maybe we have written but wish to check that everything is correct. Another advantage of most disassemblers is that they will display not only the assembly language mnemonics but also the machine language hex codes for these instructions. There are various uses for this facility, some of which will not be clear until later. I have already used the disassembler quite extensively to produce final dumps of the programs which are in the book. Using a disassembler is much easier than using an assembler and for this reason I intend to spend very little time on the subject.

Loading of the "ACS Software" disassembler, is the same as with the assembler — also remembering to load the 16K or 48K versions depending on your computer. It is also important to remember that it is possible to have the assembler and the disassembler in memory at the same time with both versions. This makes for a very powerful programming tool even though you may only have a 16K Spectrum. This only works providing that you load the assembler BEFORE the disassembler. Once the disassembler is loaded into the computer it is possible to execute this routine by one of the following commands:

48K Spectrum; RANDOMIZE USR 54ØØØ,
16K Spectrum; RANDOMIZE USR 266ØØ.

As soon as one or other of these commands has been executed then the words STARTING ADDRESS? are shown at the top of the screen. Now it is a simple matter to enter the address in decimal from which you wish the program to disassemble. If you make a mistake whilst typing in this address, do not use "Delete", instead, press key "E". As soon as you have successfully entered the starting address the first page of disassembled machine language code is displayed on the screen. If you wish to continue disassembly then simply type "C"; if you wish to go back to the "Starting Address?" statement, then type "R". If you wish to make a copy of the present screen onto the printer then type "P", and if you wish to exit the disassembler and go back into Basic, then type "E".

One of the most useful features of both the assembler and the disassembler is that they can remain in memory whilst you are operating your own BASIC or machine code program, as long as the machine code program is not located at the same addresses as the assembler or disassembler, and would not be affected by a NEW statement. Surely, the NEW statement clears the memory totally? Fortunately this is not true and, as you may remember from earlier, the NEW command only sets back to nought all the memory from the beginning of RAM to RAMTOP. This is specifically so that such things as user defined graphics, and/or machine code, can be put safely

where they cannot be corrupted by a BASIC program, or by the NEW instruction.

How do we set this RAMTOP to a specific address? Simply type CLEAR xxxxx (where xxxxx stands for RAMTOP). This means that all memory after RAMTOP is protected. For example, to protect all the memory from 28ØØØ upwards, so that our renumbering program, plus the assembler and disassembler are in memory and incorruptible, then we would type: CLEAR 27999. Remembering that the upper limit of RAMTOP is always used, it is necessary to set RAMTOP to 1 less than the address from which we want protection.

You will find that many programs have the CLEAR instruction in the BASIC listing, so that protection is there. It is also interesting to note that if you exit from the disassembler, you will see that in the short BASIC program already in memory there is a CLEAR statement. Also remember that although CLEAR does blank the screen, it does not clear any of the memory above RAMTOP.

BREAKING INTO MACHINE CODE

The way in which the BREAK key works in BASIC is very simple, the computer simply scans the keyboard, and if the BREAK key has been pressed then it simply jumps out of whatever it is doing at the time and prints up an error message, then jumps back to BASIC. Unfortunately we cannot do that directly in machine code, because the facility just does not exist in hardware. We can get around this problem by using your own short BREAK routine, which can be stored with the other machine code routines, and which can be accessed every so often in order to check for the BREAK key. The short machine code routine below does this. Not all instructions used in this program have been covered so far and in fact some of them, particularly the RRA instruction, will not be mentioned. Suffice to know that if the BREAK key has been pressed then this program will cause an immediate return to BASIC:

```
ld a,127
in a,(254)
rra
ret nc
```

At the moment it is only possible to put the above as part of our machine code routine, but later on we will learn how to have this as a simple sub-routine of our main routine. This provides a very useful and powerful tool. Later on I shall also talk about how to use this facility not only to detect whether the BREAK key has been pressed, but also whether any other specific key has been pressed. Let's carry on then.

MONITORS

The question you have doubtless already asked yourself is what do I do if I haven't got an assembler, do not wish to get one, or simply cannot afford one? There is no simple way to get all the advantages of an assembler without purchasing such a program, but it is still possible to enter machine code into memory, though only in its machine language code form, with some of the advantages offered by an assembler — we use a MONITOR. There are also some advantages which are not inherent in assemblers that make up a little for the fact that it is necessary to convert the assembly language mnemonics into machine language codes ready for entry in the "assembly to machine language codes" conversion tables at the back of this book. Such conversion tables are also available from other sources.

DEBUGGING PROGRAMS

You will find that even though you have meticulously checked a particular machine code routine, there are bound to be bugs occurring every so often. They creep in everywhere, whatever you do! To recover from a crash in machine code is not possible and it is therefore particularly important that you find the errors in a program before you actually execute. For this reason, and other reasons which will become apparent, a very great asset is to have a debugging program. These are

available in packages, just as the assembler and disassembler. Again, for your information, the debugging program that I have used throughout this book, and shall use as a specific example now and later, is not produced by the same software house as the disassembler and assembler, although they should have one available by the time this book is published. The one used here is available from ARTIC Computing.

Loading in this Spectrum debugging program is as simple as ever, being only necessary to type LOAD "", but do remember to load the correct version for the machine you are using. Once the debugging program has loaded, you have various options open to you, but because this program is designed for use on your own machine code routines it would seem sensible to first load in the machine code Pixel Scroll routine which was saved earlier in this chapter. Although there are no bugs in this program (they have already been removed!), it will still act as a very useful example. Additionally, because it is not possible to load your own machine code routines which have not been saved via this debugging program, it is necessary first to exit from the debugging program back into BASIC before you load. This is done simply by pressing the "X" key followed by ENTER. Once you have done this you can load in the machine code routine normally. Then, to re-enter the machine code debugging program, type PRINT USR 30884 for a 16K Spectrum, or PRINT USR 63652 if you have a 48K machine. It is very important that you do get these numbers correct, (they are written on the instructions which come with the debugging program). Note that all the functions are listed in the instructions for this program, but they are only written out for reference purposes rather than for explanation. I do not intend to run through all the commands and options available on this debugging program, but let us have a quick look at those which are more important:

"Z" enables the disassembling of a short area of RAM — as does the disassembler that we looked at earlier, although the disassembler within the debugging program is not quite so sophisticated. To disassemble the area in which our Pixel

Scroll routine resides, simply type: Z 753Ø. If for any reason you wish to stop the disassembly, hit the BREAK key. Always remember to type ENTER after having typed the instructions for the debugging program. It is so easy to forget.

Note: All the values used in this debugging program are in hexadecimal, which means that there are a maximum of four figures, and decimal numbers are not allowed. For this reason it may be useful just quickly to look back to earlier chapters and quickly revise how conversions are made between decimal and hexadecimal. For your assistance however, a list of all the binary, decimal, and hexadecimal values from Ø to 255 are included at Appendix A which may be of use to you when converting. It is also possible to execute a machine code routine from the debugging program by the command G followed by the address, in hexadecimal, at which the machine code routine starts, followed by ENTER. It is then possible actually to display the contents of the main registers by pressing "D" followed by ENTER. This will show, at the top of the screen the values for all the major registers.

This is a useful facility at the end of a program, but would it not also be useful if you could discover the contents of the main registers part-way through the program? This can quite easily be done by the use of what is known as a break point which is sometimes referred to as a "quit point". All it means is that when the program arrives at a certain address then it will return straight back into the debugging program. From there you can display the registers, and, as you will find out a little later on, also display the contents of the flags. To set a quit or break point simply type: "Q", followed by the address (in hexadecimal) at which the break point is required. Then execute your machine code routine by the command shown above ('G'). As soon as the routine jumps back into the debugging program, the break point is exterminated.

Now try disassembling the program, deciding where to put break points, inserting them, running the program, and displaying the main registers' values at each break point. Also,

by use of the "F" command followed by ENTER, display the contents of the flags. There are also other less spectacular commands within ARTIC's Spectrum debugging program which I shall quickly mention. It is possible:

1. To enter a message into a certain part of the memory, simply by typing in the message via the keyboard, which will then convert this into the appropriate hexadecimal value.

2. To search a block of memory for a particular value, the address of which is then displayed.

3. To copy one block of memory into to another.

4. To display the exchange registers, which we have yet to look into.

5. To command the debugging program to replace all the occurrences of one specific value with another value.

6. To make the debugging program load and save programs written with it.

7. To modify the contents of various bytes of memory, and also enter the machine language codes, into addresses, and then run them.

8. To set specific registers with specific values, so that you can simulate certain occurrences within a program.

9. To print up the basic character codes of your routine, so that it is possible to enter them into a REM statement.

WHICH ONE?

If it comes to the crunch and you can only afford to buy one of these programs, it will probably serve you best to buy the debugging program first. Not only because it helps you tremendously when it comes to finding errors in program, but also because it is a very good aid to writing programs, is an aid to understanding how a program works and how the computer works in general. If you decide to go further in machine code, and wish to write longer and more complicated programs, then it is very likely that an assembler would be the most useful tool, as long as it is coupled with a disassembler. Next would come a debugging program.

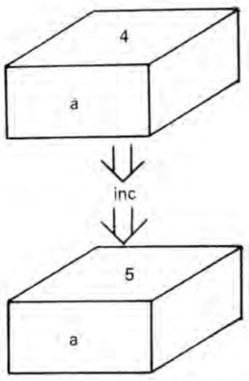
AND FINALLY....

Before going on to writing our own programs, which we shall be doing in the next chapter, I shall quickly run through four very useful instructions which have either been used already, but you may not yet fully understand, or have been alluded to rather than fully explained. As with everything that we have learned so far, it will become far clearer when actually put into practice. For this reason, the next chapter is solely taken up with the writing of machine code programs. A great deal of attention is also given over to careful description of each instruction and procedure necessary to produce a good program. The instructions that we shall be looking at now are: INC, DEC, NOP, RET.

INC:

INC is another of those instructions which comes in two forms. It can either be used on a single register or a register pair. One of the major advantages of this command is that it can be used on any of the registers. INC X will cause the contents of register "X" to be increased by one. Therefore, for example, if the A register holds the value of 5, and the INC A command is executed, then the value of register A will become 6. The effect will be exactly the same whether it is done on the A register, or the B, C, D, E, H, or L registers. With addition, if the value of A or any other register for that matter is 255, and it is "INCremented", then the value of the register will go back to zero. But now for the catch. If the value of a register is changed from 255 to Ø by the increment instruction, then the carry flag will not be set as it would have been with the ADD instruction. This is a crucial fact to remember when, later on in this book, you will want to check the value of a register to decide on the subsequent course of your program.

The diagram below is a simple representation of the action of this instruction.



If you wished to increment the contents of a register in BASIC, you would use the instruction let B = B + 1, when the register to be incremented is B. It is quite apparent from this example, that the machine code equivalent is considerably shorter, hence saving memory, and is quite definitely faster. The assembly language mnemonics and their equivalent hex codes are shown below:

INC A	3C
INC B	Ø4
INC C	ØC
INC D	14
INC E	1C
INC H	24
INC L	2C

It is also possible to Increment the value of a register-pair just as with a single register — The only difference being that this instruction will not in any way affect any of the flags including the carry flag.

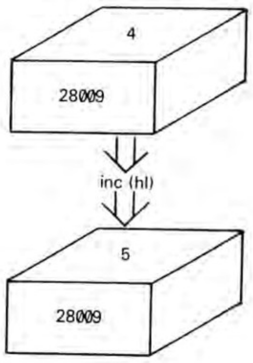
The assembly mnemonics and the hexadecimal codes for these instructions are shown below:

INC BC	Ø3
INC DE	13
INC HL	23

The operation of such an instruction is not very difficult to comprehend, though you may have some difficulty in understanding it at first. Just in case you do have any problems, here is a short machine code program to illustrate its function. By executing this program from BASIC (using the PRINT statement), the final value of register pair BC will be shown. It is interesting to play around with the contents of BC in the first statement, noting how various changes affect the answer.

```
org 30000
30000 01  00 00        ld bc,0000
30003 03               inc bc
30004 C9               ret
```

There is still one function of the INCrement statement at which we have not yet looked. It is actually possible to increment the contents of a particular location. For example, if you first load into the register pair HL the address of the location whose contents are to be incremented, and then execute the instruction "INC (HL)", the contents of the address HL will be increased by one – THE VALUE OF HL ITSELF WILL NOT BE CHANGED. The diagram below shows this a little more clearly.



The hexadecimal machine language code for this instruction is 34. Do remember that this instruction can only be used in conjunction with the HL regiseter.

Now enter the program listed below, execute it, then check the value of location 28Ø12. Re-run the routine and check again the contents of location 28Ø12. You will find that it has increased by 1. Note that the operation of incrementing the value of a location works only on that particular location, hence the maximum value which can be incremented is 255 – that being the maximum content of any location, as by now you well know.

Here is the program:

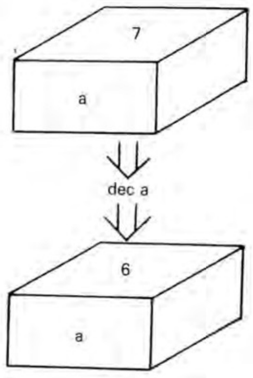
```
org 30000
30000 21  6C 6D        ld hl,28012
30003 23               inc(hl)
30004 C9               ret
```

DEC:

This instruction is exactly similar to INC, except that the value of the register or location is decremented rather than incremented – it goes down by one rather than up! For example, if we decide to do the DEC A command when A already equals 7, then the result will be that 1 is subtracted from the value of register A, the result therefore being 6. The diagram overleaf shows clearly this process.

Below is a list of the assembly language mnemonics and hexadecimal machine language codes for these instructions:

DEC A	3D
DEC B	Ø5
DEC C	ØD
DEC D	15
DEC E	1D
DEC H	25
DEC L	2D



You should by now be able to convert the example program used earlier to demonstrate both INC and DEC commands. In case you still find this confusing, here is the routine altered appropriately:

```
org 30000
30000 01 00 00        ld bc,0000
30003 0B              dec bc
30004 C9              ret
```

Again, the carry flag is not altered when the decrement instruction is executed on either a single-byte register, or a register-pair. The assembly language mnemonics and the hexadecimal codes for the decement instructions, using the register-pairs, are shown below:

DEC BC	ØB
DEC DE	1B
DEC HL	2B

As you can see from the above mnemonics and hexadecimal codes, the hex codes do bear some relationship to the assembly language mnemonics. For example, all the

decrementing instructions on the register-pairs have a B as the second digit. It is useful to remember small tips like this — it makes life easier if you do not have an assembler. Take a few minutes off to look through the book to see if you can find any other similar relationships.

As with the increment instruction, it is possible to decrement the contents of a particular location when the address of this location is held in register pair HL. Below is an example of this in use.

```
org 30000
30000 21 6C 6D        ld hl,28012
30003 2B              dec(hl)
30004 C9              ret
```

After executing this routine, check the value of location 28Ø12 by PEEKing it from basic, then re-execute this routine and re-examine the contents of location 28Ø12. In all cases, except if the previous value of that location was Ø, the contents will have decreased by 1. If, on the other hand, the content of that location is already Ø, then it will become 255 when 1 is subtracted.

RET:

By this time you will almost certainly have grasped that this instruction causes the routine to finish, and the control of the computer to go back to BASIC. The RET instruction at the end of a program is always necessary, unless you use one of the minor variations which will be dealt with later. They do not affect what we are dealing with at the moment. The hex code for the assembly language mnemonic "RET" is an easy one to remember and you will probably find that within the next few pages it will become second-nature to you — it is C9. The reason for this eventual familiarity will be your continued use of the code.

NOP:

This is a very easy command to understand — it does absolutely NOTHING. When the computer comes across this instruction, whose code is 0, it will do absolutely nothing, carrying on to the next instruction and leaving the contents of all registers, all addresses and all the flags exactly as they were. It may seem strange that anyone would want to include an instruction which apparently does absolutely nothing, whereas in fact it is very useful. This results from it often being important to be able to leave areas within a routine where nothing is held. This means that later on you are able to insert extra commands. This practice is similar to the way in which you might leave a few spare lines between each program line in BASIC. In other words, you would never number your program lines 1, 2, 3, 4, and so leave yourself no room to manoeuvre. Similarly, it can be most useful to have a command available that ensures, when required, that the computer cannot operate.

Until now we have concentrated very much on the theoretical side of machine code although I have used examples and have included some programs for your use. However, actually using machine code is its main attraction. The following chapter will therefore concentrate on the process of constructing the 'idea' of a program, coding it, and then finally debugging, running and saving it. Before we leave this chapter here is a complementary program to that used earlier. This is "Pixel Scroll Up". Use either an assembler, or the basic monitor, or the debugging program.

Have Fun!

```
org 30000
30000 21 00 40       ld hl,16384
30003 E5             push hl
30004 11 00 5B       ld de,23296
30007 01 20 00       ld bc,32
30010 ED B0          ldir
30012 E1             pop hl
HL=TOP ADDRESS OF TOP BLOCK
B=NO OF BLOCKS
30013 06 03          ld b,3
A
30015 C5             push bc
30016 E5             push hl
30017 06 08          ld b,8
B
```

```
30019 C5             push bc
30020 E5             push hl
30021 06 07          ld b,7
C
30023 C5             push bc


30024 E5             push hl
30025 E5             push hl
30026 D1             pop de
30027 24             inc h
30028 01 20 00       ld bc,32
30031 ED B0          ldir
30033 E1             pop hl
30034 24             inc h
30035 C1             pop bc
30036 10 F1          djnz,C
30038 E5             push hl
30039 11 E0 06       ld de,1760
30042 ED 52          sbc hl,de
30044 D1             pop de
30045 01 20 00       ld bc,32
30048 ED B0          ldir
30050 E1             pop hl
30051 11 20 00       ld de,32
30054 19             add hl,de
30055 C1             pop bc


30056 10 D9          djnz,B
30058 11 E0 06       ld de,1760
30061 19             add hl,de
30062 E5             push hl
30063 11 20 00       ld de,32
30066 19             add hl,de
30067 D1             pop de
30068 01 20 00       ld bc,32
30071 ED B0          ldir
30073 E1             pop hl
30074 11 00 08       ld de,2048
30077 19             add hl,de
30078 C1             pop bc
30079 10 BE          djnz,A
30081 11 20 00       ld de,32
30084 ED 52          sbc hl,de
30086 EB             ex de,hl
30087 21 00 5B       ld hl,23296
30090 06 20          ld b,32
D


30092 7E             ld a,(hl)
30093 00             nop
LD A,0 for no wrap
30094 12             ld (de),a
30095 13             inc de
```

```
30096 23          inc hl
30097 10 F9       djnz,D
30099 C9          ret
```

Do You Know How To Use It?

Chapter 7
THE WRITING OF A PROGRAM

We have now progressed a long way in learning how to use machine code and the instructions involved. It is a good time now to start looking at how we can put together our own machine code programs. By virtue of only a very few commands the following are within our competence, and so can be put to good use:

1. Add registers together
2. Add to registers
3. Subtract one register from another
4. Subtract numbers from registers
5. Increment a register
6. Increment the value of a location
7. Decrement a register
8. Decrement the value of a location
9. Load a register with a value
10. Load a register with a value of another register
11. Load a register with the contents of a particular location
12. Load a particular location with the value of a register
13. Return to BASIC and do absolutely nothing.

With this grounding, we can put together quite a comprehensive machine code routine. However, now is also the time when you find that it is not quite as easy as writing in BASIC — but just persevere and you will reap considerable benefit — you cannot simply sit down at the keyboard and write. Instead you must go about it logically and systematically; you must decide stage by stage what you wish to do and how you wish to do it. Always make sure that you have not mde any mistakes since it is not possible to recover from a crash in machine code. Another important factor to remember is that the machine code program is not as easy to go back into (to change) as is a BASIC program. Therefore an almost invaluable aid is to have flow charts and notes on exactly how you have written your program. What follows immediately is an outline of the process of "mapping out" and then the construction of my programs. You will come to have your own personal way of tackling this process, but for the moment benefit from studying mine:

1. THE IDEA OF BRIEF:

Decide precisely what you wish the program or routine to accomplish when completed. This will involve looking very carefully at the problem which you wish to solve or the ideas you wish to put into effect, and deciding then exactly what you actually want the computer to do. Write it down and use it to refer to as you go along.

2. THE OUTLINE FLOW CHART:

This ensures that you have broken down your over-all concept into its component parts. Life is a lot easier when it comes to working on each separate stage. At this time it is not necessary to go into any great detail of what you want each part to do, only the order in which you want the separate parts of your idea to be executed.

3. WORKING OUT THE STRUCTURE:

This entails going through the whole program, stage by stage, analysing what movement or operation you wish the computer

to undertake at each particular point. Bear in mind the range of commands which are at your disposal. From this it is a natural progression to the next step.

4. THE FINAL FLOW CHART:

This basically is an amalgamation of each of the particular stages in the workings of this routine, put sequentially and logically into an easily understood form.

5. CONVERSION:

This does not mean changing assembly language into machine code, but rather the Basic operation of transforming what you have written in the form of short statements into assembly language instructions. These can then be put into the "assembler understood" version i.e. the mnemonics. Now put the instructions into a logical order, just as you will require the computer to execute them.

6. THE DRY RUN:

Now work through the program step by step, not using the computer, but with pen and paper — recording the contents of the registers used, the contents of any particular addresses which might be altered, and the state of the flags themselves. If all this goes well you are ready for entry into the assembler, but if not, then at least you have not spent time converting to machine code, typing in and running before discovering the problem.

7. ENTERING IT:

To be familiar with the procedure for using your own particular assembler is very important and is why I have elected to stay with one particular program throughout the book.

8. CHECKING:

Before going any further it is worth just checking, either by disassembling or thorough examination, that you have not made any minor mistakes whilst entering it into the assembler.

Often mistakes are detected by the assembler, but equally as often they are not and then cause a completely different instruction to be assembled. This can mean the whole program crashes without it being evident whas has gone wrong.

9. DEBUGGING:

This can be helped by a debugging program, the use of which I have already outlined. Debugging a program whilst it is in memory is not only invaluable for checking for errors within your program, but also helps to give you a much better understanding of what is going on within the computer.

10. SAVING:

As I have already pointed out, the saving of machine code is not the same operation as saving a Basic program. It is therefore important to be able to do both, so as to be able to make a permanent store of your routines.

11. RUNNING IT:

You are now at the stage when everything is "A OK" and ready to execute your routine. Even this is not the easiest and most straightforward operation.

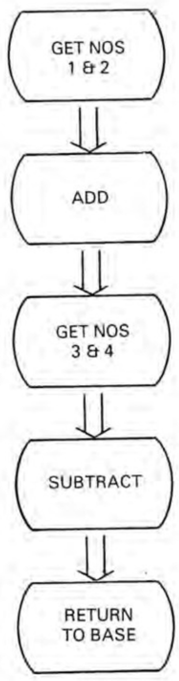
Now let us start this process with our own idea, so that we can follow it through to a final program by the end of this chapter.

THE IDEA:

What we are going to do is to write a machine code routine which is:

1. accessible via BASIC
2. is able to communicate with BASIC
3. can add two 16 bit numbers together
4. can subtract one 16 bit number from another, leaving the results available for use from the BASIC program

THE OUTLINE FLOW CHART:



WORKING OUT THE STRUCTURE:

The first problem that we come up against is how to assign values from BASIC into the machine code program, so that the 16 bit numbers to be added or subtracted are controlled via BASIC. Remember, we cannot simply assign a variable in BASIC and expect recognition by the machine code. It is a complicated procedure to transfer the contents of a BASIC variable into a machine code register. For this reason I feel that it would not be a particularly viable solution. Remember also that it is not possible to load a register from BASIC, as there are no register manipulating instructions within BASIC. Fortunately, there is one operation which can be done via BASIC or via machine code. This is transferring a number into a particular location, and then transferring that value out of the location into either a register or a variable as in BASIC. If we want to load a particular number into a location in BASIC, then we use the POKE command. To transfer it from that location into a register we use the LOAD location instruction. All we are really doing is to put a value into a box, go into machine code, and then take it out again.

The next problem is transferring the answers from machine code back into a form accessible via BASIC. In this case there are two different methods of solving the problem. Because of what was said above, it would seem totally fruitless to investigate the possibility of accessing a machine code register in BASIC.

The first solution that may be used is very similar to the way in which we are transferring into the machine code program in the first instance. In other words, load a particular location with the answer whilst still in machine code, and then take the answer out again when back into BASIC.

On the other hand, if the PRINT USR xxxxx, instruction is used, where xxxxx is the location of the machine code program, then the contents of the register-pair BC will be printed when the machine code routine has finished. This means that if we make

sure that the answer is in register-pair BC then all we have to do is use the PRINT statement. It would work equally as well if we used the "LET A = USR xxxxx" instruction, again where xxxxx equals the address of the machine code program, because the contents of the register-pair BC will be transferred to the variable A. This solution is very useful indeed, but its major drawback is that you can only store one answer in the register-pair BC. Rather than return to BASIC and then have to re-enter the machine code, it is advisable to use a combination of the two solutions. In other words, we would store the first answer in addresses which can then be assessed via BASIC, and have the second result loaded into BC, so it can be printed or transferred to a variable.

Having now decided exactly how we are going to transfer the constants from BASIC to machine code, and the way in which we are going to transfer the answers from machine code back into to BASIC, we must now work out the sequence of occurrences between these two operations. What we are doing now is to decide whether to do the addition or the subtraction routines first. It is a matter of no consequence, just as long as we remember to put the result of the first routine into a memory location and the result of the second routine in the register-pair. As a matter of course I have decided to do the addition before the subtraction within our program. This decision was not arrived at after excessive analytic processes, I just tossed a coin.

Flow Chart — Version 2: Below is the second version of the original flow chart; look at it carefully in the light of the above:

Get the first two values out of their particular memory addresses.

Add these two numbers.

Store the result in a box, or location.

Get the second two values out of their respective memory locations.

Subtract one from the other.

Store the result in register-pair BC.

Return to BASIC.

NOW FOR THE BASIC

Store the four values

Run the machine code routine

Print the result of the second routine

Retrieve and display the result of the first routines which were loaded into the memory locations earlier on within the machine code routine

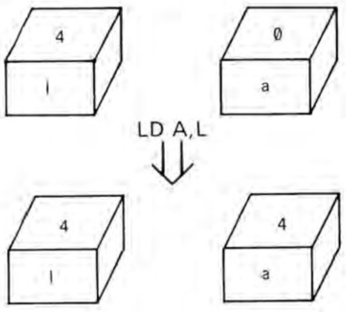
Stop

HOW DO WE ADD TWO 16-BIT NUMBERS?

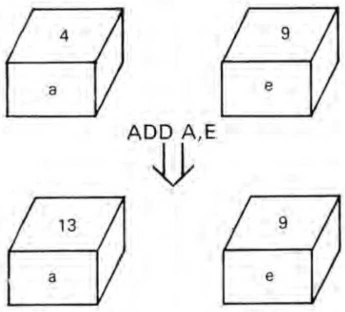
Let us presume that the numbers which have been accessed from the computer memory have been loaded into the registers HL and DE. The result of the addition is then to be stored in register pair HL. It is important to remember that H is the high byte part of register pair HL, and L is the low byte part of HL. Similarly, D is the high byte part of register pair DE, and E the low byte part. As in ordinary addition, we add first the "low bytes" — to do otherwise would prove more than problematic, a fact which will become more obvious as you proceed.

We need therefore to add register E to register L. Unfortunately, as you may remember, it is only possible to add other registers to register A. So how do we get out of this? Simple. All we have to do is transfer the value of L into register A, then add register E to register A. The result of adding E and L together is now held in register A, so that the answer in A can be transferred back into L.

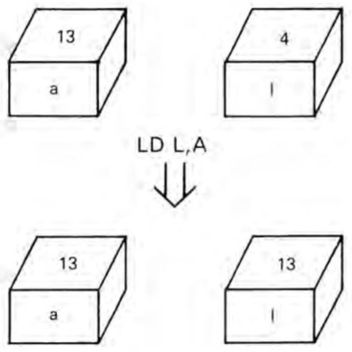
We have now decided on our first two instructions for our machine code program. First we must "transfer L to A", this is shown in the diagram below:



We must then add register E to register A; again this is shown diagrammatically below:



We now have the result in register A, but that is not where we want it!! We want the whole result to be in register-pair HL, so we must put the result of the addition of the low bytes into register L. This can be done by a simple transfer of the contents of A to L.

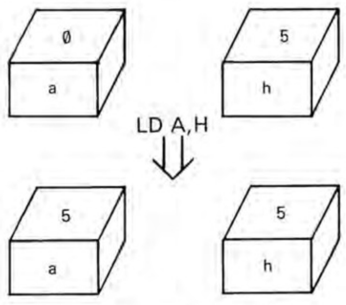


We should now take a quick look at what happens if adding the two bytes together results in a number greater than 255 — as 255 is the largest number which can be held in any single register. One good thing about microprocessors is that the response to this occurrence is wholly consistent and predictable. Everything goes back to zero! Adding 1 to 255 starts it back at Ø; adding 2Ø to 255 sends it back through zero to 19. Additionally, the carry flag is set to 1. The carry flag is a single bit within the F register to which the CPU itself refers in order to ascertain whether or not a number has overflowed. If a number has gone over the 255 barrier then there has definitely been an overflow, and the carry flag is set to 1. If, of course, there is no overflow, then the carry flag will re-set to Ø. To refer to this from time to time is absolutely essential — its importance being hard to overstate.

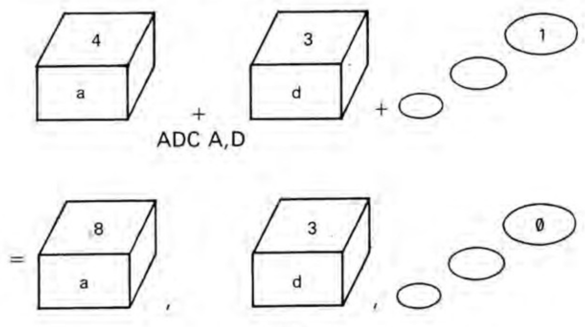
The carry flag itself is a strange beast: it cannot be accessed directly, and so it is not possible to load A, for example, with the status of the carry flag. On the other hand there are instructions which take the carry flag into account. The status of the carry flag can be controlled however, and so it can be set and re-set at will. It is one of those commands which take the carry flag into account which we shall use when adding the high bytes

together. We shall use an instruction which does the following: adds E to H plus the status of carry flag. The status of the carry flag being set by the addition of the low bytes, and not changed by any transfer of registers, this will effect the correct addition.

Back to the task in hand. We were at the stage when the low bytes had been added together and we were ready to add the high bytes. Given that we cannot add D to H, but only D to A, we transfer H to A. E.g.

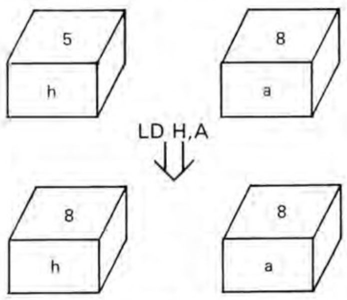


Now we add register D to register A and take into account the status of the carry flag. E.g.



The next instruction therefore is to "add D and carry to A".

We now have the result of D plus H plus the carry flag in the A register, and we want to have it back in the H register so that the final result is held in HL. The transfer is effected thus:

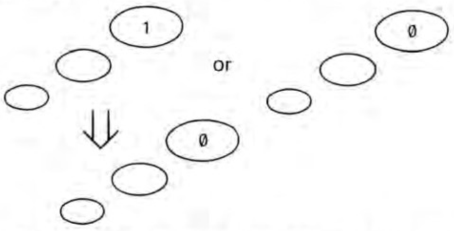


"THE RESULT IS NOW HELD IN HL (At long last!)"

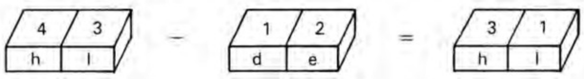
The "addition" is complete, but it does now have to be made accessible via BASIC. We can either put the value of HL into BC, so that it can be printed on the screen, or be transferred into a BASIC variable, or we can store the result in memory locations, all of which can be accessed via BASIC. Of these options — using BC to hold the result — would seem to suit our purposes best. However, as we are likely to want to make use of the BC register during the subtraction part of the program so we will have to use memory locations for storage. Two memory locations are needed for this two byte sum (16 bit). The locations at which the answer may be loaded can be anywhere in RAM, but first make sure that you are not overwriting some other program. If you are in any doubt whatsoever, then refer back to the section on the memory map. For our purposes addresses 27004 and 27005 will be fine. It is possible to split the answer up into two addresses which are not adjacent to one other, but it is rather pointless and makes the programming rather difficult. We can now instruct the computer to store HL in locations 27004 and 27005.

NOTE:
The computer stores the low byte before the high byte, so do remember to multiply the second address contents by 256, rather than the first.

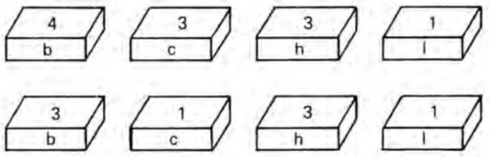
Now to deal with subtracting one 16-bit register from another, using a register-pair subtraction command. HOW? Take as a starting point that the two required values have been entered into the register-pairs DE and HL, the object being to subtract the value of DE from the value of HL — the result to be left in HL. Only one command is needed to accomplish this, but remember that this command also brings in the carry flag. If the last instruction executed has caused the status of the carry flag to become 1, then our result may be wrong, so we do have to make sure that the carry flag status is 0. This is indirectly possible. For now, make do with the instruction, "Reset carry flag to 0", e.g.



With the carry flag at 0 it will not affect the result of subtraction. We can go ahead and subtract (with carry) DE from HL — shown diagrammatically as below:



You have effectively subtracted the value of DE from HL, leaving the result in HL. All there is to do now is to decide on how to transfer this result back into a form which can be accessed via the BASIC. Earlier we avoided using the BC register-pair in case it was needed during the construction of the program — perhaps a laborious way of making a point, but one worth making nonetheless. Now that it is evident that "BC" is free to be used it can come into its own. Transfer H to B, transfer L to C. The result which was originally held in HL is now also held in BC. See below:



The way in which our program is to work is now firmly under control, and so the next stage is to transfer the original values from Basic into memory, and then have the machine code program handle the transfer of the contents of each location into the correct register pairs.

Transferring Initial Values Whilst In BASIC For Use In Machine Code.

The first requirement is to convert the four 16-bit numbers into pairs of 8-bit numbers; these can be placed in memory and then recalled by the machine code. Working in hexadecimal this process is relatively easy. Take the first two digits and find their equivalent in "base 256", then take the final two digits and find their equivalent in "base 256". The result provides the two halves of the 16-bit number. At the back of this book you will find a table of all the binary, hexadecimal, and decimal numbers from 0 to 255. Do not hesitate to refer to the tables — they will take some of the labour out of the exercise. Remember that the computer itself will save the low byte value first and then the high byte. Do the same. Whilst it will not alter the result, just as long as the relevant changes are made in the program, it will help towards 'getting in the habit at this stage'.

The eight 8-bit numbers can be stored in any part of RAM, as

long as they do not interfere with anything else. I have already chosen to use 27000 as my starting point. For the addition:

```
L = 27000
H = 27001
E = 27002
D = 27003
```

and for the subtraction:

```
L = 27006
H = 27007
E = 27008
D = 27009
```

The result of the addition will be held in addresses 27004 and 27005.

Having converted the 16-bit numbers into two 8-bit numbers it is an easy job to put the numbers into the addresses in preparation for running the machine code routine.

HOW DO WE GET INTO MACHINE CODE?

It is not possible to GOTO the machine code routine or GOSUB the routine as both these instructions are for BASIC programs use only. BASIC does have a very special command totally for the machine code user — USR — which stands for User Sub Routine. Within the context of BASIC, USR is not a complete command, and so it is necessary to prefix it by one of the following:

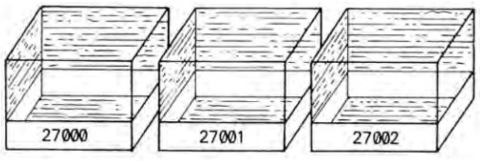
PRINT, hence you print to the screen the value return in the register pair BC,

LET, which means that you can assign a variable the value of the contents of register pair BC on a return, and

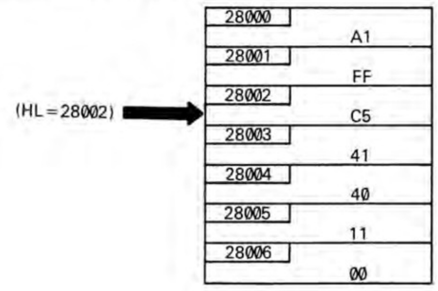
RANDOMISE, which simply sets the random seed to the result in register pair BC.

Both PRINT and LET are wasteful of memory, and for this reason RANDOMISE is generally used as the accessing command for a USR routine. A USR routine is very much like a GO SUB routine, in that it is a sub routine; to get out of it again include the instruction RETURN, which in the context of machine code has been shortened to RET.

Next we have to look at how to access the constants which have been stored via BASIC. The best way of representing the numbers stored in their locations is by way of a diagram:



On the face of it, the next move should perhaps be to use a command to load the contents of address 27000, for instance, into register C. Be assured, this would involve using four lengthy routines, and is therefore definitely to be discouraged. Instead we make use of the facility for loading a register with the contents of a location whose address is held in HL. Using this method, and increasing or decreasing the value of HL, in order to select which bytes to load into the registers, is referred to as "using the HL register as a pointer". The operation of a pointer is clarified in the diagram below:



Now to put this into effect.

First set the pointer to 27000; "load HL with 27000"

Get first half of first 16 bit number into C; transfer contents of location HL to C

Move pointer on one location; "increase value of HL by 1"

Get second half of first 16 bit number into B; "transfer contents of HL to B"

Move pointer on one location; "increase value of HL by one"

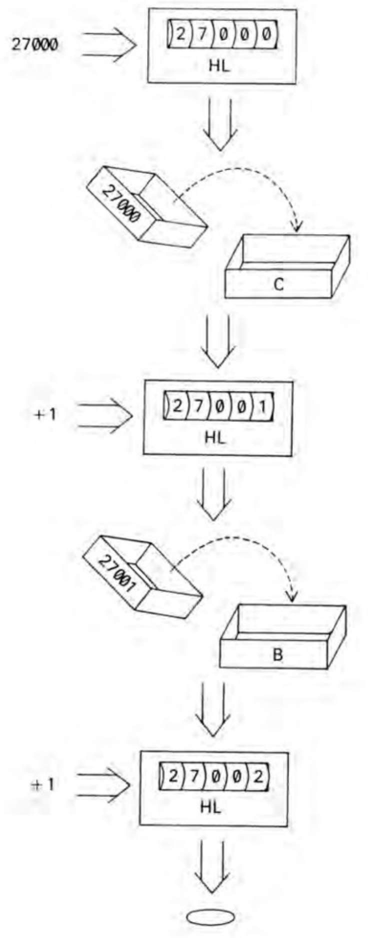
Transfer first half of second 16 bit number to E; "transfer contents of HL to E"

Move pointer on one location; "increase value of HL by 1"

Transfer second half of second 16 bit number to D; "transfer contents of location HL to D"

The transfer is now complete, but the only problem is that we have to load one of the values into BC because HL was used as the pointer. HL is the only register pair which can be used as a pointer, as it is the only register which may hold a location to be accessed. Do the following two instructions: "transfer B to H", "transfer C to L". The constants which were stored from BASIC are now in their correct machine code register-pairs.

Next is the transfer of the constant from BASIC to machine code for the subtraction section. Notice the great advantages of the pointer system. One of the nicest things about this system is the ease of having the computer access different addresses simply by changing the initial contents of the HL register — which is acting as the pointer. Another aspect of the pointer system is that it is possible not only to access one location after the other — going forwards, but also to access one location after another going backwards (by decreasing the value of the pointer). This is accomplished by the decrement instruction. This will become clearer in a step by step plan of how to put the values from memory into their register locations.



Load pointer with top value: "Load HL with 27009"

Transfer second half of second 16 bit to D: "Transfer contents of location HL to D"

Move pointer back one location: "Decrease value of HL by 1"

Transfer first half of second 16 bit number to E "transfer contents of location HL to E"

Move pointer back 1 location: "Decrease value of HL by 1"

Transfer second half of first 16 bit number to B: "transfer contents of location HL to B"

Move pointer back 1 location: "decrease value of HL by 1"

Transfer first half of first 16 bit number to C: "transfer contents of location HL to C"

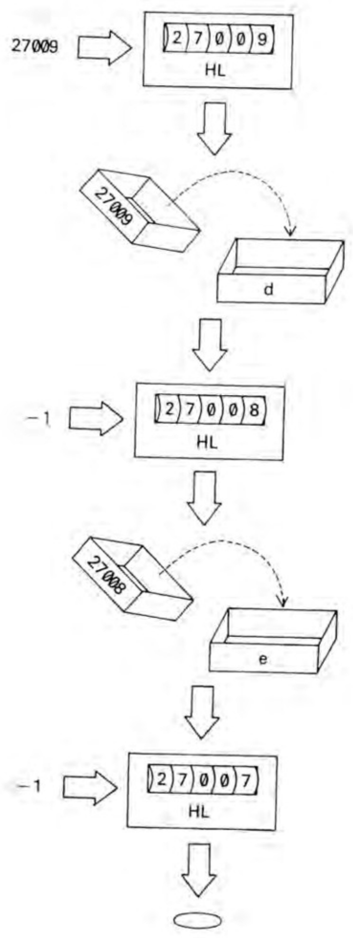
We now have the same problem as before: we have to transfer register pair BC into HL — BC was used to store the values on a temporary basis as HL was being used as the pointer. So:

"transfer B to HL"
"transfer C to L"

The diagram below shows exactly what is happening.

The process of transferring the data from BASIC to machine code is now complete. They have been processed and re-stored in a way which is accessible via BASIC. Therefore the actual writing of the program is almost completed. All that remains is to transfer control from machine code back into BASIC.

The USR command which we use to transfer control from BASIC to machine code has already been referred to and likened to the GOSUB command in BASIC. The line pointer must now be moved from its present line to the assigned



subroutine. When the subroutine is at an end a RETURN command is included to tell the computer to go back to one line after the initial GOSUB location instruction. For example, if all the lines are numbered in increments of 10, and the GOSUB instruction is on line 10, then the RETURN instruction will cause the program to continue operation from line 20. In machine code, the same instruction "Return" is used to transfer control from the present machine code instruction, back to the line after the original USR location instruction in BASIC. With a USR instruction at line 10, and the next line being line 20, when the machine code routine has finished and encountered a RET instruction, the Basic program will continue to run from line 20.

Therefore, the final instruction of a machine code program will be to "RETURN to Basic".

Putting This All Together

Good practice dictates that, before going further, a clear and logical listing should be made of the instructions that make up the program.

The first section shows the BASIC required to transfer the original values; the second section shows the actual workings of the machine code program, including transfer in and out, and the addition and subtraction routines. The final section shows how to access the answers from BASIC.

BASIC:

Transfer first addition value into locations 27000 and 27001 (L H)

Transfer second addition value into locations 27002 and 27003 (E D)

Transfer first subtraction value into locations 27005 and 27006 (L H)

Transfer second subtraction value into locations 27007 and 27008 (E D)

Jump into machine code subroutine

MACHINE CODE:

Load register pair HL, with 27ØØØ

Transfer contents of location HL to C

Increase value of HL by 1

Transfer contents of location HL to B

Increase value of HL by 1

Transfer contents of location HL to E

Increase value of HL by 1

Transfer contents of location HL to D

Transfer B to H

Transfer C to L

Transfer L to A

Add E to A

Transfer A to L

Transfer H to A

Add D and carry to A

Transfer A to H

Transfer HL to locations 27ØØ4 and 27ØØ5

No operation

No operation

Load register pair HL with 27ØØ9

Transfer contents of location HL to D

Decrease value of HL by 1

Transfer contents of location HL to E

Decrease value of HL by 1

Transfer contents of location HL to B

Decrease value of HL by 1

Transfer contents of location HL to C

Transfer B to H

Transfer C to L

Reset carry flag to Ø

Subtract with carry register pair DE from register pair HL

Transfer H to B

Transfer L to C

Return to BASIC

BASIC:

Print final value of register pair BC (subtraction result).

Get the result of the addition by PEEKing and PRINT.

CONVERSION TO ASSEMBLY LANGUAGE

Our expectations of the computer are now clear, and we are in a perfect position to convert our instructions into assembly language mnemonics, which can then be entered into an assembler. It is also a good idea at this point to find out the hexadecimal code for the instructions, which can then also be entered via the BASIC monitor or a debugging program. This also serves as a worthwhile check. Below is a full table showing the instruction, the assembly language mnemonic and the hexadecimal code:

Worded Instructions	**Assembly Language**	**Hex Codes**
Load HL with 27ØØØ	LD HL,27ØØØ	21 78 69
Contents of location HL to C	LD C,(HL)	4E
Increase value of HL by 1	INC HL	23
Contents of location HL to B	LD B,(HL)	46
Increase value of HL by 1	INC HL	23
Contents of location HL to E	LD E,(HL)	5E

Increase value of HL by 1	INC HL	23
Contents of HL to D	LD D,(HL)	56
Transfer B to H	LD H,B	60
Transfer C to L	LD L,C	69
Transfer L to A	LD A,L	7D
Add E to A	ADD A,E	83
Transfer A to L	LD L,A	6F
Transfer H to A	LD A,H	7F
Add D and carry to A	ADC A,D	8A
Transfer A to H	LD H,A	67
HL to locations 27004 and 27005	LD(27004),HL	22 7C 69
No operation	NOP	00
No operation	NOP	00
Load HL with 27009	LD HL,27009	21 81 69
Contents of location HL to D	LD D,(HL)	56
Decrease value of HL by 1	DEC HL	2B
Contents of location HL to E	LD E,(HL)	5E
Decrease value of HL by 1	DEC HL	2B
Contents of HL to B	LD B,(HL)	46
Decrease value of HL by 1	DEC HL	2B
Contents of location HL to C	LD C,(HL)	4E
Transfer B to H	LD H,B	60
Transfer C to L	LD L,C	69
Reset carry flag to 0	AND 0	E6 00
Subtract with carry DE from HL	SBC HL,DE	ED 52
Transfer H to B	LD B,H	44
Transfer L to C	LD C,L	4D
Return to Basic	RET	C9

DRY RUNNING IT

Now is the time to check the program carefully for any errors that might have crept in along the way. If there is a major error then it is likely that the computer will "crash" without giving you any basis on which to determine just what has gone wrong. Having gone to all the trouble of typing in the program after is not the best of times to discover a 'bug'. This is the stage of the game when it is good general practice to do a "dry-run".

What is meant by "dry running" a program is, in effect, running the program on paper! You might, for example, use a 'table' of all the registers, inserting their values after each instruction — keeping note of the status of the Carry Flag etc. Errors usually show up quickly using this method. Another useful feature of

this method of checking, is that it creates a document which shows what the program is actually doing, and so allows for subsequent verification or alteration. Included here is a simple table that will assist in dry-running your own programs. Consider each assembly language mnemonic, decide what it does, and then insert into the table the result of the instruction. For example if the command LD A,40 is encountered, then enter 40 in the A column. Used in this way it will provide a "ready reckoner" that will allow for immediate verification of register contents. Expanding the table to take in all the memory locations so far used in your program may help to familiarise you with its use.

H	L	D	E	B	C	A	(Carry) if known

ENTERING IT INTO THE ASSEMBLER

Whilst the hexadecimal codes have been listed, it is not usual even to work them out if an assembler is being used as this would be doing the job of the assembler. We now have the assembler ready, and also the program which we want to assemble.

Once the assembler is LOADed its operating instructions should be given. In this case a "GO" command, to establish that there is a program waiting to be assembled. The "ORG" instruction is then used, together with a location address, to identify where the assembled program is to be held. Though ready now to enter the assembly language mnemonics after the REM statement, it is worth while putting a comment at the beginning of this program for identification at a later stage if necessary. This is done by putting an exclamation mark after the REM statement followed by the comment. In this case it would be appropriate to write "addition and subtraction routine".

Each mnemonic must be carefully entered to make sure that no errors occur. Check the final list of mnemonics against the original list. It is possible to have more than one instruction per line, by separating them with a semicolon, but it is much easier to read off the routine if only one instruction per line is used. This is what the list of instructions should look like when they have been entered into the assembler:

FINAL VERSION

```
ld hl,27ØØØ                    217869
ld c,(hl)                          4E
inc hl                             23
ld b,(hl)                          46
inc hl                             23
ld e,(hl)                          5E
inc hl                             23
ld d,(hl)                          56
ld h,b                             6Ø
ld l,c                             69
ld a,l                             7D
add a,e                            83
ld l,a                             6F
ld a,h                             7C
adc a,d                            8A
ld h,a                             67
ld (27ØØ4),hl                  227C69
nop                                ØØ
nop                                ØØ
ld hl,27ØØ9                    218169
ld d,(hl)                          56
dec hl                             2B
ld e,(hl)                          5E
```

```
dec hl                             2B
ld b,(hl)                          46
dec hl                             2B
ld c,(hl)                          4E
ld h,b                             6Ø
ld l,c                             69
and Ø                            E6ØØ
sbc hl,de                        ED52
ld b,h                             44
ld c,l                             4D
ret                                C9
```

We are now almost ready to assemble the routine, but there are still things to do and point out.

Put a comment on the next-to-last line to indicate that this is the end of the routine. The "Go", "Finish" and "ORG" instructions are very important to the assembler itself, and so it will refuse to operate if any of these are absent. The assembly will cease and an error code will be displayed. If too many spaces, or insufficient spaces are input, then it may well misinterpret. This can cause a major problem as it is a difficult error to detect.

Once the program has been assembled it is advisable to make a copy on cassette or microdrive so that if the program should crash then all would not be lost. To save the assembled program type:

SAVE "name" CODE xxxxx, yyyyy

where xxxxx stands for the location from which you wish the routine to be saved and yyyyy being the number of bytes in this routine. The instruction CODE is the vital part, telling the computer to save machine code. To verify simply type VERIFY "" CODE. To save the BASIC program which holds the mnemonics, which is easier to change and then recompile, treat as though it was a normal program. I.e.: SAVE "name".

To load in the machine code program in its hexadecimal form, type LOAD "name" CODE. This will load in a machine code program called "name", and will locate it at the address from which it was saved. To reload a normal BASIC program i.e. the program which holds the assembly language mnemonics, type LOAD "name".

The BASIC program used to hold the mnemonics before assembly commenced has now outlived its usefulness, and may be erased. Do not try to do this via the NEW command — not unless you want to lose everything — use CLEAR xxxxx (where xxxxx is the location from which you wish to 'protect', minus one). In this way RAM TOP is reset and the BASIC program can now be NEWed. You can be assured that your machine code is safe and sound, as is the assembler which still resides in memory.

Now for the control program which will organise the transferring of variables into memory, out again and print the results on the screen.

```
10  POKE 27000,4:POKE 27001,1:
    POKE 27002,3:POKE 27003,2
20  POKE 27006,7:POKE 27007,1:
    POKE 27008,8:POKE 27009,9
30  PRINT "SUB =";USR 28000
40  PRINT "ADDITION=";PEEK 27004 +
    (PEEK 27005)*256
50  STOP
```

To have the control program, in BASIC, loading in at almost the same time as the machine code program is more convenient. This is done by making the BASIC program auto-RUN with the first line of this program responsible for loading in the machine code program. They can then be saved next to each other on cassette.

After verifying a copy of the program, only running remains. Except to run it. This is effected by a GOTO or RUN instruction to the BASIC program at the right line, because the PRINT USR 28000 instruction is embedded within the BASIC program. Then maybe experiment with different values in the registers and observe (predict?) the results.

Jumping Is Fun

Chapter 8
JUMPING ABOUT

Developing the "right technique" has been central to everything so far covered. Now is the time to extend the repertoire of commands available for use in programming — certainly this is necessary if your creative instincts are ever to be given full rein. The more that is known, the more can be used. It will become increasingly apparent that many permutations can be derived from only a very few commands, thus making for great versatility. This chapter will focus on Simple Jumps, Conditional Jumps, Relative Jumps, Conditional Relative Jumps, Calls and Conditional Calls. Additionally, account will be taken of the PC Register, the RET instruction, and other associated aspects of the computer's inner workings.

THE PC REGISTER

The organisation of the CPU's registers has already been covered in general terms, and some particular attention given to those which are more easily manipulated (HL, BC, etc.,). There are other Register-Pairs which either cannot be directly accessed, or can only be accessed for relatively basic operations. This results from the CPU's need to reserve space for storing its own variables where they are accessible as well as being protected from outside interference. The PC register is a suitable example. This register-pair holds the address of the location in memory of the current command being executed and therefore acts as a pointer to the CPU giving it the location for the start of the next instruction.

When the computer is first turned on the value of the PC register is 0000, so the first command the computer immediately executes is that which is situated at the first location in memory. This is why the Sinclair ROM is mapped on to the first 16K of memory. In other words, via the use of a ROM chip, the values of locations 0 to 16383 are predetermined and for this reason whenever the computer is turned on it immediately goes into what is known as the "start up" routine. Once in a machine code routine the PC register always contains the location of the current instruction, so that when the instruction is completed, PC changes according to the length of the instruction. The location of the next instruction is therefore known. It can be presumed then that, by externally altering the value within the PC register, it is possible to change the location from which the CPU will get its next instruction can be determined.

The first problem to overcome is that there are no commands which will directly alter the contents of the PC register as there are for the HL or BC registers for example. Fortunately there is an instruction which changes the value within the PC register, namely the JUMP or JP instruction. It acts very much like the GOTO instruction within BASIC, simply by loading the PC register with the required value — the location from which the next instruction will be taken. In BASIC, if at line 100, and requiring the next instruction to be taken from line 1050, then the instruction GOTO 1050 would be used at line 100. In machine code the operation is exactly the same. It follows that when we have a short routine from locations 30000 to 30012, and when this routine has been completely executed and we wish to start executing a routine at locations 32000 onwards, the process will look like this:

```
30000 — 30012      : first routine
30013 JP 32000     : jump to locations 32000
32000 —            : second routine
```

An example of this instruction is shown below:

```
org 30000
30000 3E 00            ld a,0
30002 06 07            ld b,7
30004 C3 00 7D         jp 32000
```

```
32007 60               ld h,b
32008 6F               ld l,a
32009 C9               ret
```

Whilst it is undoubtedly useful to be able to jump from one part of memory to another, and execute instructions from that part of memory, in how many cases is it necessary to write a routine in two different parts of memory? Would it not be far more useful to be able to execute the command from another part of memory only if a certain situation is true or false? For example, in a BASIC program with a long and complicated calculation, the computer is required to indicate where the result is higher or lower than expected and a routine similar to the following would be employed:

```
500   IF answer is greater than predicted THEN GOTO 1000
510   IF answer is less than or equal to predicted THEN GOTO
      1500
1000  PRINT "greater than"
1010  STOP
1500  PRINT "less than or equal to"
1510  STOP
```

Notice that in this BASIC program the statement STOP has been used to end the routine and there is no parallel instruction in machine code — not least because a total cessation is never required. When concluding a machine code routine return to BASIC is always required. In normal use the RET instruction returns the control of the computer back to BASIC.

A list of the six most commonly used conditional jump instructions are shown below:

JP z	:jump if zero
JP nz	:jump if not zero
JP c	:jump if carry flag set
JP nc	:jump if carry flag not set
JP m	:jump if minus
JP p	:jump if positive

JP z means that the next instruction will only be taken from the prescribed location when the zero flag is set, hence the result of the last calculation was zero. If the result was not zero, and this condition is false, then the instruction from the next location in memory will be taken.

For example:

```
org 28000
28000 3E 00          ld a,0
28002 C6 00          add 0
28004 CA 6B 6D       jp z,28011
28007 3D             dec a
28008 C3 62 6D       jp 28002
28011 C9             ret
```

In this program the test is to determine whether or not adding zero to the accumulator results in zero. If it does, then the control of the program will go back to BASIC. If not, then the value of the accumulator will be decremented; i.e. it will equal itself minus 1, and then will test again whether adding zero to it results in 0, and carry on like this until the accumulator equals zero. Ensure that the instruction used before a conditional jump does in fact alter the state of that particular flag. When in doubt, check the table at the back of the book.

Jp nz, means that the next instruction will only be taken from the assigned address if the zero flag is not set, hence the result of the last calculation was not zero. If, as in the previous example, the preceding instruction was "add zero", followed by jp nz, 32000 then the machine code would like like this:

```
29999 3D              dec a
30000 FD 69           add 0
30002 CA "end"        jp nz,29999
30005 C3 "32000"      jp 32000
```

The BASIC equivalent would be:

IF A plus 0 equals 0 THEN GOTO 32000

JP c, means "jump only if the carry flag is set". In other words,

should the last calculation have caused an overflow then the carry flag will be set and the condition will be met. A simple example would be to instruct that where registers A and B, in combination, are greater than 255 then jump to new address, but if not, then continue. A program to do this is shown below:

```
org 30000
30000 80             add a,b
30001 DA 30 75       jp c,30000
```

The BASIC equivalent woulds be: IF A plus B is greater than 255 THEN GOTO 30000. This provides an easy check for the possibility of errors occurring when the sum of two variables is a number too great for the CPU register to handle.

The JP nc instruction is very similar except that you will only jump if the carry flag is not set. This tests whether or not the result of the last calculation has an overflow; if not — the values added together being less than 256 — then it will jump to the assigned location. An example is shown below:

```
org 30000
30000 80             add a,b
30001 D2 30 75       jp nc,30000
```

The BASIC equivalent would be:

IF A + B < 256 THEN GOTO 30000.

The final two conditions that we have available to us for a simple jump instruction are minus or positive. They are concerned with whether the sign flag reflects a positive or negative value. As we have yet to look closely at this flag, suffice it to say that JP P means, "jump if sign flag reflects a positive number". The JP M command means, "jump if sign flag reflects a negative number".

Throughout these examples, actual addresses have been used in order to demonstrate precisely what is happening, but using an assembler, or writing a program which can be put anywhere in memory, it is much easier to use "labels". To go to a routine once, and then back to the beginning again, label the beginning of the program START, at the end execute the instruction JP START. Using an assembler there is no need to use direct addresses as the assembler will do this for you, working out the addresses for each part of the program. Labelling particular bytes within the program which may be used as storage space for answers or calculations during the routine is also possible. This function has to be one of the most useful, apart of course from converting the mnemonics themselves. A short example of how labels can be used within programs is shown below. Do not be concerned if the way in which the program works seems rather obscure, just take note of the labels and how they are used and assigned.

1. Basic listing for ACS Assembler.

```
   1 REM go
   2 REM org 30000
  10 REM ld hl,22527
  20 REM ld b,192
  30 REM !CHANGE B & HL FOR DIFF
ERENT BLOCKS OF SCREEN"
  40 REM A;ld c,32
  50 REM B;ld e,(hl)
  60 REM rl (hl)
  70 REM dec hl;dec c
  80 REM jr nz,B
  90 REM bit 7,e
 100 REM jr z,C
 110 REM push hl;ld de,32
 120 REM add hl,de
 130 REM set 0,(hl)
 140 REM !RES 0,(HL) FOR NO WRAP
 AROUND
 150 REM pop hl
 160 REM C;and a
 170 REM djnz,A
 180 REM ret
 190 REM fi
```

2. Assembled listing.

```
org 30000
30000 21 FF 57     ld hl,22527
30003 06 C0        ld b,192
CHANGE B & HL FOR DIFFERENT
BLOCKS OF SCREEN
A
30005 0E 20        ld c,32
```

```
B
30007 5E           ld e,(hl)
30008 CB 16        rl (hl)
30010 2B           dec hl
30011 0D           dec c
30012 20 F9        jr nz,B
30014 CB 7B        bit 7,e
30016 28 08        jr z,C
30018 E5           push hl
30019 11 20 00     ld de,32
30022 19           add hl,de
30023 CB C6        set 0,(hl)
RES 0,(HL)FOR NO WRAP AROUND
30025 E1           pop hl
C
30026 A7           and a
30027 10 E8        djnz,A
30029 C9           ret
```

To jump to a location earlier in the program is of course possible. Say at location 30ØØØ, and needing to execute the routine at 28ØØØ, the instruction JP 28ØØØ at location 30ØØØ is totally legitimate. There is one very important factor that has to be guarded against when including jumps which loop back on themselves and that is the use of unconditional jumps that do not have RETurn instructions between them. This is the quickest way of getting the computer to go into an eternal loop — with no hope of escape! Breaking out of such a loop in BASIC requires only the pressing of the Break key, but in machine code there is no such facility. Some computers have the facility for self resetting so that you can start all over again without losing the routines within memory, but this is not possible with the ZX80, ZX81, or ZX Spectrum. The only way to get out of an eternal loop within machine code is to 'pull the plug out'. Whilst in machine code the Spectrum will not register the pressing of the Break key, or any other key for that matter, unless of course a key-scan routine is written into the program.

We have now looked at direct jumps which allow us to load the PC register, or the Program Counter, with a new address in order externally to alter the location from which the next instruction will be taken. There is another type of jump instruction — useful when writing short routines which are not dependent upon the actual area within memory that they are located.

This new type of jump is known as the RELATIVE JUMP. The reason for this title is that it allows you to jump to a location "plus or minus a number relative to the current location". Instead of jumping to, say, location 28ØØØ when at location 28ØØ5, a Relative Jump of minus 5 can be used. This may seem a rather complicated way of jumping from one location to another, but it does have the advantage that the routine may be relocated anywhere within memory. In using relative jumps, because there are no actual addresses used for the JUMP instruction, the routine can be moved from 28ØØØ to 24ØØØ or any other location within memory, without the need for conversion.

This facility is particularly useful when the routine may be used more than once within a larger program, or when converting a program for use on a larger computer. Similarly, it can come into its own when there is insufficient room within memory for a routine, and so is necessary to move the routine down to make room for the remainder. The Jump Relative command can save a great deal of work later on. There is one major limitation to this command, it is only possible to jump to a location a maximum of 'plus 129 bytes' or 'minus 126 bytes' relative to the current address. This may seem rather too limiting, but in practice 'plus 129 bytes and minus 126 bytes' usually provides more than ample space within a standard routine. Calculating displacements greater than this is tricky. Suffice to say therefore that the Jump Relative instruction (JR) is best regarded as a local instruction.

Calculating the displacement for a relative jump is reasonably easy, though it does require some care. To use negative numbers within a machine code program is not directly possible. For a Jump Relative displacement which is positive, i.e. between Ø and 129, the actual number which must be inputted to the computer is that of the displacement. Where the displacement is negative, i.e. between Ø and minus 126,

then the number which must be put into the computer will be 256 plus the negative displacement. Using the assembler employed for this book it is possible simply to type the displacement, positive or negative and in this way it will not be necessary to worry about working out the actual displacement.

When executing a jump relative to the instruction, the computer is being instructed to go to an instruction a designated number of bytes greater than that which it would normally. In other words if the computer executed the instruction JR Ø (jump relative zero bytes), then this would have no effect, because the computer would in any case be intending to execute the command at the next address. This would simply be increasing the value of PC by Ø. However, because the length of the Jump Relative instruction is 2 bytes, by executing the instruction JR − 2 it will effectively be re-executing the JR − 2 instruction, which results in an infinite loop. The only way out of it would be to pull the plug.

Below is an example of the use of the Jump Relative command, and also how useful it is when combined with the use of labels within an assembler.

```
org 30000
Start
30000 3E 29          ld a,41
30002 06 28          ld b,40
30004 18 06          jr Part2
30006 00             nop
30007 00             nop
30008 00             nop
30009 00             nop
30010 00             nop
30011 00             nop
Part2
30012 0E 0E          ld c,14
30014 81             add c
30016 18 02          jr +2
30018 18 EC          jr Start
30020 C9             ret
```

It is interesting to note that at no time during this program are actual addresses used; the actual routine may be located anywhere within the computer's RAM without alteration.

It is also possible to have Conditional Relative Jumps in exactly the same way as Conditional Jumps. The only difference being that there are fewer Conditional Relative types available for use and only the following are possible.

JR Z	: jump relative if zero flag set
JR NZ	: jump relative if zero flag not set
JR C	: jump relative if carry flag is set
JR NC	: jump relative if carry flag not set

These can be used in exactly the same manner as the relative versions of the jump instruction, simply by remembering to use a displacement rather than a direct address.

Within BASIC, as well as having the GOTO instruction, we also have the GOSUB instruction. This allows us to execute a routine other than the main program, and then return back to the main program without the need for another GOTO instruction. The implementing of short sub-routines is therefore made easy. One advantage is that it is possible to re-use a particular routine many times within a program. For example, to carry out a specific operation requiring a number of commands, three or four times within a program would not necessitate building into the main program more than once, but to have the operation as a subroutine. The sub-routine could be either before or after the main program, and could be accessed by a GOSUB command in BASIC. When completely executed, the RETURN instruction at the end of the routine would cause the control of the computer to return to the statement after the GOSUB in the main program. To call a machine code sub-routine from a machine code program requires the instruction "CALL", followed by the address (or label if you are using an assembler), of the sub-routine.

An example of how this can be used within a program is shown below:

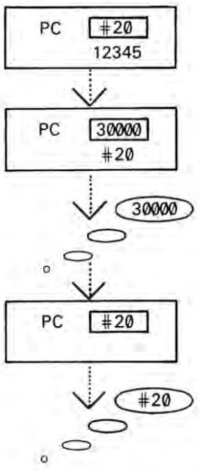
```
10 RANDOMIZE USR 30000
```

```
org 30000
start
30000 CD 6E 6D        call,28014
30003 C9              ret

org 28014
subrt
28014 3E 00           ld a,0
28016 06 00           ld b,0
28018 C9              ret

20 PRINT "ROUTINE COMPLETE"
```

Another use of the CALL instruction is to execute sub-routines already situated within the ROM chip. Though many of the routines within the ROM are very well interwoven with the execution of BASIC, and other facilities best left to the ROM, there are still very many which can be very useful. Some of these routines are short when the only reason to use the CALL instruction is to save time copying them out into the RAM and our programs. Many however are long and complicated, which makes copying them time-consuming and rather wasteful in terms of memory space. The use of the CALL instruction does in addition eradicate the chances of error.

Once a sub-routine has been executed it is necessary to return to the main program. In BASIC this is done using the RETURN instruction, in machine code by using the RET instruction. This may seem rather confusing, as so far the RET instruction has been used to return control of the computer back into BASIC at the end of a machine code routine. The fact is that the RET instruction does a very simple job by always returning control to the position where the GOSUB or CALL instruction has been used. Hence, where a machine code routine was executed via the BASIC USR instruction, and a RET instruction was encountered, then the control would return to BASIC. Alternatively had a CALL instruction been executed in machine code, and at the end of the sub-routine you wished to revert to the main program, the RET instruction would cause this to happen.

Once a CALL has been made, the location from which it came is stored by the computer so that when the RETurn instruction is encountered then the computer is able to go back to the correct place in the memory. When the RETurn instruction has been encountered and control returned to the main program, the value of the location stored by the computer is replaced by the previous value. In other words, when going into a machine code program via the BASIC USR instruction, the address held by the computer is that of the next BASIC instruction. When going into a machine code sub-routine from machine code, via a CALL instruction, the address of the CALL will also be held by the CPU. Whilst in this sub-routine if the RET instruction is encountered it will go back to the address held by the computer. In that case it would be the main machine code program. Because this sub-routine has now been completed, the value held by the computer is that which was held before the sub-routine was called. This would be the address of the subsequent BASIC instruction. The diagram shows rather more clearly what is actually happening.



It is also possible to have Conditional Call instructions, just as with jumps and relative jumps. They do in fact work in exactly the same way as those already explained above, and I do not intend to stress the point further. With CALL instructions the use of labels again dramatically increases the ease with which they can be used in a program, and assist in the creation of a structured program. To divide a program into a series of sub-routines becomes possible and then they can be accessed via a short master-program. Working this way makes for a more understandable listing when finished, and also makes it very easy to adjust and expand the program. Regrettably there are no such things as Relative Calls, and so it is necessary to use direct addressing — making the result not relocatable. Even so, relocating a program is not particularly difficult because, as you will remember, it is not necessary to remember an address to return to after a sub-routine is completed. The sub-routine can be located within the memory, and then the master-program itself relocated. The mnemonics and hexadecimal and decimal codes for these instructions are listed at Appendix B.

How To Get On (Or Off) With A ZX Spectrum

Chapter 9
USING THE SCREEN AND KEYBOARD

This chapter will primarily be about how the computer and its operator interrelate or "communicate". The main focus therefore will be the 'keyboard' and the 'screen'. We will start off by quickly looking at the arrangement of the keyboard and at the way in which we can use the system variables as an easy method to read the keyboard. You will then be left with some practical examples and routines which can be included in your programs. We will look at the way in which the screen is arranged and how it is mapped in the memory. As the screen is mapped into a large proportion of the Random Access Memory within the computer we are able easily to alter the contents of the screen, though Clive Sinclair has not made life any too easy for programmers by arranging the screen in a somewhat peculiar fashion.

Attention will be given to how the Spectrum's inherent difficulties might be overcome, how to perform such tasks as printing Sinclair characters onto the screen and how to define new characters and subsequently display these on the screen. Finally there will be a machine code routine which allows plotting of any point onto the screen and because of its amazing speed this routine should prove to be far more useful than might at first appear. The keyboard is effectively split into rows and columns, which enables the computer to identify the pressing of any key with the minimum of instruction. This also allows the computer to register when two keys have been pressed together, provided they are in separate rows. Examples of this are the Caps Shift and Symbol Shift keys. Were they next to each other and in the same row, then this would confuse the computer which would be unable to differentiate between these and certain other keys pressed at the same time.

The way in which the computer actually reads the keyboard is rather involved, though it is useful and interesting to understand the basic concept of how the keyboard is arranged. Fortunately for us, we do not need to understand or even use the routine within ROM which scans the keyboard, because of two very useful variables which the computer keeps constantly within the system variables area. This ranges from locations 23552 to 23665. Various variables are held including current colours, length of beep, and most importantly for us, the last key pressed. By peeking location 23557 it is possible to find out whether or not a key has been pressed. To prove this, type in the short BASIC program shown below:

```
  1Ø IF PEEK 23557=5 THEN PRINT
AT Ø,Ø;"Key pressed"
  2Ø IF PEEK 23557<>5 THEN PRINT
 AT Ø,Ø;"            "
  3Ø GO TO 1Ø
```

This program ascertains whether or not there is a 5 at location 23557 and if so it prints to screen "Key Pressed". Take your finger off the key, so the contents of location 23557 no longer equal 5 and it will blank out the words "Key Pressed" with spaces. The GOTO 1Ø statement at line 3Ø simply means that the routine carries on endlessly.

CODE		CHR$	CODE		CHR$
32	=		33	=	!
34	=	"	35	=	#
36	=	$	37	=	%
38	=	&	39	=	'
40	=	(	41	=	)
42	=	*	43	=	+
44	=	,	45	=	-
46	=	.	47	=	/
48	=	Ø	49	=	1
50	=	2	51	=	3

CODE		CHR$	CODE		CHR$
52	=	4	53	=	5
54	=	6	55	=	7
56	=	8	57	=	9
58	=	:	59	=	;
60	=	<	61	=	=
62	=	>	63	=	?
64	=	@	65	=	A
66	=	B	67	=	C
68	=	D	69	=	E
70	=	F	71	=	G
72	=	H	73	=	I
74	=	J	75	=	K
76	=	L	77	=	M
78	=	N	79	=	O
80	=	P	81	=	Q
82	=	R	83	=	S
84	=	T	85	=	U
86	=	V	87	=	W
88	=	X	89	=	Y
90	=	Z	91	=	[
92	=	\	93	=	]
94	=	↑	95	=	_
96	=	£	97	=	a
98	=	b	99	=	c
100	=	d	101	=	e
102	=	f	103	=	g
104	=	h	105	=	i
106	=	j	107	=	k
108	=	l	109	=	m
110	=	n	111	=	o
112	=	p	113	=	q
114	=	r	115	=	s
116	=	t	117	=	u
118	=	v	119	=	w
120	=	x	121	=	y
122	=	z	123	=	{
124	=	|	125	=	}
126	=	~	127	=	©
128	=		129	=	[illegible]
130	=	▪	131	=	■
132	=	▚	133	=	[illegible]
134	=	[illegible]	135	=	[illegible]
136	=	[illegible]	137	=	[illegible]
138	=	[illegible]	139	=	[illegible]
140	=	[illegible]	141	=	[illegible]
142	=	[illegible]	143	=	[illegible]
144	=	A	145	=	B
146	=	C	147	=	D
148	=	E	149	=	F
150	=	G	151	=	H
152	=	I	153	=	J
154	=	K	155	=	L

CODE		CHR$	CODE		CHR$
156	=	M	157	=	N
158	=	O	159	=	P
150	=	Q	161	=	R
152	=	S	163	=	T
154	=	U			

This information is very useful, but it does not identify which key has been pressed which is done by peeking location 23560. This location is often labelled "Last K", because it holds the value of the last key pressed and it does, in fact, carry on holding the number, even when the key is not pressed. Use this in conjunction with the contents of location 23557 to first find out if a key has been pressed, and then which key has been pressed — always finding out first whether a key has been pressed. Below is a short example program written in BASIC to show what happens if only the contents of location 23560 are used; i.e., the value of the last key pressed:

```
10 PRINT PEEK 23560,: GO TO 10
```

Try this and you will find that it is of little use on its own, but now try the program below, which uses both values in conjunction with each other.

```
  10 IF PEEK 23557=5 THEN PRINT
AT 0,0;CHR$ (PEEK 23560)
  20 GO TO 10
```

Note that to print the actual character pressed, it is necessary to use the CHR$ instruction on the contents of the location. The computer only stores the actual CODE value of the key in location 23560. The computer uses only a single byte to hold the code of the particular character, and this therefore allows for a maximum of 256 different characters to be available.

This information can now be used to write a machine code routine which works in a very similar way to the BASIC one above, but which returns the value of the last key pressed in register C — the only difference being that it will only return if a key has been pressed. We now come to a very effective use of

the Conditional Jump Relative instruction. Caution though — as we are in machine code, and because we are reading each key separately, the computer will not register the BREAK key. With no key pressed then we will forever stay in machine code. This situation might be quite alright if we know that a key is to be pressed, and that we are going to go straight back into BASIC anyway after that key is pressed — in which case the BASIC BREAK key will be operable. However should we use this within a totally machine code program it would be advisable to build-in a function whereby, when the break key is pressed, then an "L' message and a return to BASIC would ensue.

Consider now how to force an error message. The Spectrum is quite capable of delivering understandable error messages, and it is in fact not too difficult to utilise these and generate them via our own program, even though they are not directly applicable to machine code. Forcing an error message in this way is not particularly difficult, especially when you know how, but might be rather confusing at this stage. Instead, use a method which relates to the BASIC rather than to the machine code. Remember that should you try to print the code for the first 32 characters, an error results. The reason for this is that these characters are used by the computer for instructions specific to itself and they may be concerned with colour, or the end of a line. As you know, colour is embedded into a program simply by pressing caps shift, symbol shift and then one of the numbers. All you are doing is embedding one of these codes into the program and then an error message ensues. Why not try it; type PRINT CHR$ (13). This operation will result in an error message. It is easy to use this fact within our machine code program by loading the register C with a value less than 32 and then returning to BASIC.

The short machine code program below uses this property to work — and to stay short:

```
        KEY-P EQU 23557
        LASTK EQU 23560
```

```
START   LD A, (KEY-P)
        SUB 5
        JR NZ, START
BREAK?  LD A, (LAST K)
        SUB 127
        JR Z, EXIT
        LD C, (LAST K)
        LD B, Ø
        RET
EXIT    LD C, 13
        LD B, Ø
        RET
```

The program starts by labelling the two locations 23557 and 2356Ø, with their system variables names, KEY-P and LAST K. In this way, when referring to them during the rest of the routine, names can be used rather than the actual numbers. This makes life easier for typing and for reading back later. The actual routine starts by loading the register A with the value held by the system variable KEY-P, it then subtracts 5 from this value (because if the answer then is Ø then a key has been pressed). This is because, as you will remember, a value of 5 held by this variable indicates that a key has been pressed.

Assuming the result of register A minus the value 5 equals Ø, then a key has been pressed, and if it is not Ø then the Jump Relative Non-Zero instruction will cause the operation to go back to START and read the value of KEY-P into register A again.

In this way it goes into an eternal loop until a key is pressed and because the system variable KEY-P does not tell us which key has been pressed, we now test for the BREAK key. This is done in almost the same way as we tested for a key being pressed: by loading register A with the value of system variable LAST K, and subtracting the code of the BREAK key. If the BREAK key has been pressed then the result will be Ø, and the program will jump to the short exit routine. By now it can be seen that a key

has been pressed, but it is not the BREAK key, so now to load the value of whichever key was pressed into the C register so that it can be printed onto the screen when back in BASIC. It is important that we do make the value of register B zero because, when in BASIC we cannot differentiate between the two registers B and C, and a number other than Ø in register B would obviously cause the result to be wildly wrong.

This completed, we return to BASIC. The final part of the program is the short routine labelled EXIT, which is accessed only if the BREAK key has been pressed, and which simple loads the C register with an illegal value — loading the B register with Ø. The computer will then return itself to BASIC. As can be seen, this whole routine is completely relocatable and it would be well worth saving onto cassette for future use with either a BASIC or machine code program.

Having looked at this process, we can now see why it is not usually possible to break out of a machine code program, especially those commercially available, as authors are rarely keen on you actually being able to see what they have written. By using this method you can get yourself out of a lot of difficulties which might occur.

Now to move on to look at the computer's main outputting device — The Screen — known as an output device because it relays information 'out' to us. The screen itself is arranged in a weird and wonderful manner. 6K of memory is used to hold the actual dot-pattern on the screen, i.e., whether an actual dot is on or off the screen. A further 1K of memory is used to store the particular colours of each character and whether each character is flashing, is bright etc. In this book we shall not be dealing directly with the actual colour part of the display, but simply with hi-res graphics and character generation.

There are 256 points or pixels arranged horizontally across the screen. As they can only be either "on" or "off", the state of each group of 8 pixels can be recorded within one byte. The 8 pixels are dealt with as though they were a binary number and if the pixel is on, then it counts as a 1 in binary notation, but if it is off then it counts as a Ø in binary notation. The result is an 8 digit binary number: 1Ø1Ø1Ø1Ø

Prove this by poking the first byte of the screen memory with the binary value shown above. This is done via the instruction POKE 16384, BIN 1Ø1Ø1Ø1Ø (ENTER). Now try entering different arrangements of binary digits, but always be sure to use no more than 8 digits as this adds up to a maximum of 255 in decimal (the maximum value which the computer can handle). This arrangement applies to the whole of the screen — 6144 times. This can be illustrated by the short program below:

```
1Ø FOR a=16384 TO 16384+6143
2Ø POKE a,BIN 11111111
3Ø NEXT a
```

Try changing the value of the binary number as some quite interesting effects can be produced. You will notice that when filling the screen memory area with a particular digit, it does not cause the screen to be filled line by line sequentially — from top to bottom of the screen. Rather, the screen is split up into three distinct sections. Each of these 3 sections is split up into 8 character blocks, running from top to bottom. First the top row of each character block is filled, then the second row, continuing to the 8th and bottom row. This process then restarts for the middle block and then for the bottom row of the 3 blocks. This may seem rather strange and it certainly is!!

Another way of illustrating this is to draw a design or picture onto the screen, and then save the screen memory area, which ranges from address 16384 to address 22527, clearing the screen and then reloading. The screen memory area is fixed in a certain part of the memory, and as this is always separate from the BASIC or machine code program area, the idea of loading the screen with a design or picture at the beginning of a program is often used as an attractive introduction as it effectively uses no memory which would otherwise be used by the program. Type in the short BASIC program shown below, and then save via the command SAVE ""CODE 16384, 6144.

Clear the screen by the command CLS, and then reload the screen via the command LOAD ""CODE. The result should be quite interesting.

```
1Ø LET n=631
2Ø LET a=12Ø
25 PLOT 55,27: DRAW a,a,n*PI
```

Now for another example. Load in the 2 machine code routines for scrolling each pixel either left or right, writing a short routine which loads in the screen design just saved, and then repeatedly scrolls it left; after 1 complete rotation of the whole screen then repeatedly scrolls right. There is no need to write this in machine code. The result should demonstrate quite clearly the use of machine code routines within a BASIC program.

As you will doubtless have gathered, due to the unusual arrangement of the screen, performing such a simple task as printing a character to the screen requires quite an involved program. However, we can conclude that because the computer itself is capable of doing this, then the necessary routine must be embedded in the ROM area somewhere and we shall take advantage of this fact. The importance of this routine has ensured that it has been put very close to the beginning of the memory, starting at location 16 (decimal) — but how do we access this routine? To our benefit it has been put in a subroutine and so can be CALLed. A design feature of the Z80 is the in-built facility for accessing some of the more important and oft-used routines quickly and economically and for this purpose a set of instructions have been incorporated which apply solely to eight of the first addresses in memory. Perhaps most important of all, they are only one byte long! One such routine, the one to be used for accessing the screen printing routine, is 'RST 16'. This has the same effect as "CALL 16", and a RET instruction is therefore needed at the end of the routine — after which the control will go back to the command 'one after' RST 16.

There remain two matters to attend to before the command is executed. First the PRINT position must be set in BASIC and although this is possible in machine code, matters would be unnecessarily complicated. Second, the A register must be loaded with the code of the character to be printed. This is useful not only because of the relative ease of translation of a character to its code, using the table given earlier, but also use of User Define Graphics in machine code is allowed.

Here is a simple routine for printing the character "A" to the top left of the screen. Remember to access the machine code in the way shown (in BASIC):

```
10 PRINT AT 0,0;
20 RANDOMIZE USR 30000
```

```
        org 30000
30000 3E61      1d a,97
30002 D7        rst 16
30003 C9        ret
```

Now try changing the print position and the contents of register A (the character to be printed). Apparent from this is the ease of printing a UDG (User Defined Graphics) character by loading the A register with the appropriate code. A method is now needed whereby a UDG character can be represented in machine code. Unfortunately the routines used by the computer to do this in BASIC are not easily accessible from machine code. Additionally, because this is in any event a complicated process, I am giving you a short routine which allows UDG characters to be formed. It is not strictly necessary for you to understand how it works at this stage — just use it. Enter it and then access it by a CALL command, followed by 9 bytes of data. The first is the number of the graphics character, i.e., key A or Chr$ 144 = Ø, key B or Chr$ 145 = 1, and so on. This makes it possible to specify which character is to be defined. The other eight bytes are the 'bit arrays' which are used in the normal fashion, and so make up the defined character.

The first routine below actually defines the graphics character. The second is an example in which characters 144 and 145 (A and B keys) are defined.

```
org 30000
equ 65368 UDG
30000 0E 00          ld c,0
30002 E1             pop hl
30003 7E             ld a,(hl)
30004 A7             and a
30005 87             add a,a
30006 87             add a,a
30007 87             add a,a
30008 11  58 FF      ld de,UDG
30011 47             ld b,a
30012 7B             ld a,e
30013 80             add a,b
30014 5F             ld e,a
30015 7A             ld a,d
30016 06  00         ld b,0
30018 88             adc a,b
30019 57             ld d,a
Loop1
30020 79             ld a,c
30021 30  08         sub 8
30023 30  07         jr nc,Exit
30025 0C             inc c
30026 23             inc hl
30027 7E             ld a,(hl)
30028 12             ld (de),a
30029 13             inc de
30030 18  F4         jr Loop1
Exit
30032 23             inc hl
30033 E5             push hl
30034 C9             ret

org 28000
28000 CD 30 75       call C-Gen
defb 0
defb 1
defb 3
defb 7
defb 15
defb 31
defb 63
defb 127
defb 255
28012 CD 30 75       call C-Gen
defb 1
defb 255
defb 127
defb 63
defb 31
defb 15
defb 7
defb 3
defb 1

28024 C9             ret
```

There is no necessity for you to understand all the commands needed to write a machine code plotting program, but as it can prove to be a very handy program the routine which does this is set out below:

```
8000  00        nop
8001  00        nop
8002  3A0080    ld a,(8000)
8005  CB3F      srl a
8007  CB3F      srl a
8009  CB3F      srl a
800B  4F        ld c,a
800C  3A0180    ld a,(8001)
800F  CB3F      srl a
8011  CB3F      srl a
8013  CB3F      srl a
8015  47        ld b,a
8016  E600      and 00
8018  3E17      ld a,17
801A  98        sbc a,b
801B  D8        ret c
801C  50        ld d,b
801D  47        ld b,a
801E  7A        ld a,d
801F  CB27      sla a
8021  CB27      sla a
8023  CB27      sla a
8025  57        ld d,a
8026  3A0180    ld a,(8001)
8029  92        sub d
802A  57        ld d,a
802B  3E07      ld a,07
802D  92        sub d
802E  57        ld d,a
802F  F5        push af
8030  78        ld a,b
8031  E618      and 18
8033  F640      or 40
8035  67        ld h,a
8036  F1        pop af
8037  84        add a,h
8038  67        ld h,a
8039  78        ld a,b
803A  E607      and 07
803C  0F        rrca
803D  0F        rrca
803E  0F        rrca
803F  81        add a,c
8040  6F        ld l,a
8041  79        ld a,c
8042  CB27      sla a
8044  CB27      sla a
8046  CB27      sla a
8048  47        ld b,a
```

```
8049  3A0080    ld a,(8000)
804C  90        sub b
804D  FE00      cp 00
804F  281D      jr z,806E
8051  FE01      cp 01
8053  281C      jr z,8071
8055  FE02      cp 02
8057  281B      jr z,8074
8059  FE03      cp 03
805B  281A      jr z,8077
805D  FE04      cp 04
805F  2819      jr z,807A
8061  FE05      cp 05
8063  2818      jr z,807D
8065  FE06      cp 06
8067  2817      jr z,8080
8069  FE07      cp 07
806B  2816      jr z,8083
806D  C9        ret
806E  CBFE      set 7,(hl)
8070  C9        ret
8071  CBF6      set 6,(hl)
8073  C9        ret
8074  CBEE      set 5,(hl)
8076  C9        ret
8077  CBE6      set 4,(hl)
8079  C9        ret
807A  CBDE      set 3,(hl)
807C  C9        ret
807D  CBD6      set 2,(hl)
807F  C9        ret
8080  CBCE      set 1,(hl)
8082  C9        ret
8083  CBC6      set 0,(hl)
8085  C9        ret
8086  00        nop
8087  00        nop
8088  00        nop
8089  00        nop
808A  00        nop
808B  00        nop
808C  00        nop
808D  00        nop
808E  00        nop
808F  00        nop
8090  00        nop
8091  00        nop
8092  00        nop
8093  00        nop
```

Getting On and Off

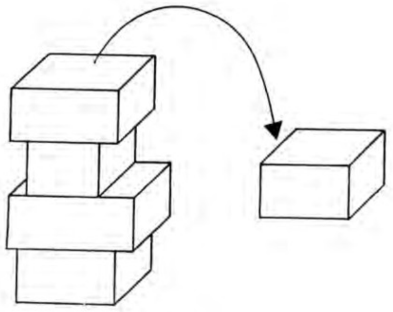
Chapter 10
THE STACK

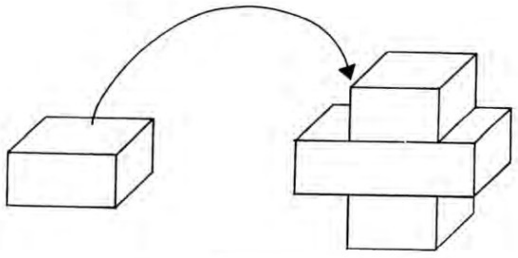
This chapter will be devoted to exploring the use of further machine code commands, some new concepts and a particularly useful area used by the CPU itself. It will be necessary to look back at the INC and DEC instructions, which will lead us onto the LDIR and LDDR instructions — a full understanding of these relies heavily on complete familiarity with the INC and DEC instructions.

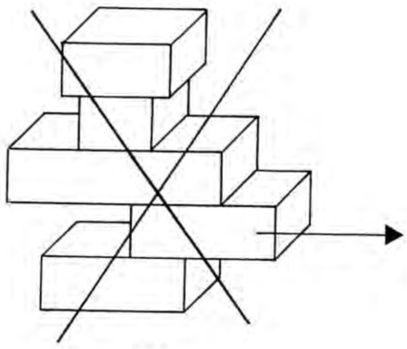
For perhaps the first time we can now deal with a term which means precisely what it says — the function of The Stack is to be a Stack and nothing but a Stack. The Stack is for stacking numbers. A number can be put on top of the Stack or taken off — always off the top, never from anywhere else.

Think of it as if it were a tall tower of boxes. It is possible to take off the top-most box, or indeed to put another on top of the existing pile, but attempt anything else at your own risk! This is exactly what happens in machine code — the only difference being that each box is numbered with an address, though one of the advantages of the stack is that it is not necessary to concern ourselves with the address.

There is never any actual restraint on the size of a stack, though the bigger it gets the less room there is for anything else, plus the fact that other code may even be overwritten. Setting up of a stack is a simple matter, deciding at which location within the memory the stack should start, then assigning the system variable STK-P to that value. This in effect creates a Stack







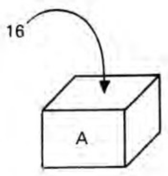
Pointer. The Stack Pointer points to the location within memory at which the next value is to be placed.

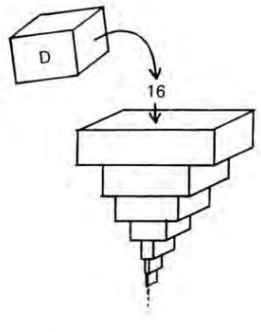
```
org 30000
equ 28000 STK-P
30000 3E 10          ld a,16
30002 FD 22 60 6D    ld (STK-P),16
30006 21 60 6D       ld hl,STK-P
30009 35             dec (hl)
```

Without an assembler the process is a little more difficult. First decide on a location which can be used to store the Stack Pointer, then make sure that the value within it is Ø, load the register pair HL with its location, and then load the location with the value of the Stack Pointer.

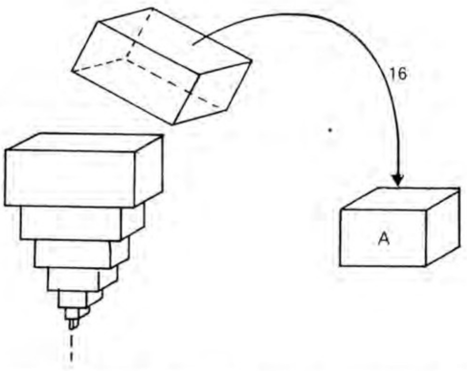
```
30010 3E 10          ld a,16
30012 21 60 6D       ld hl,28000
30015 77             ld (hl),a
30019 2B             dec hl
```

When any value is placed on top of the stack the value of the Stack Pointer must be decreased by 1, so that the next time a number is added to the stack not only will it not overwrite the first, but it will be put in a location "1 less". To put a value onto the stack first put it into the register A. This is then loaded into the location held by the Stack Pointer, and the value of the Stack Pointer is decreased by 1.





All that remains now is to work out a routine for taking a number off the stack. In this case, load the register A with the value held at the top of the stack, and then increase the value held by the Stack Pointer by 1.



This program allows us to generate and use a simple stack. The principle of having a stack of numbers, from which numbers can be taken, or to which they can be added may seem rather space-consuming, but the use of a stack in a program, using the routines above, can in practice be exceedingly useful and be an excellent way in which to simplify a program. Probably the simplest method of using a Stack, and one in which it would be easiest to read off again, is to label the two sub-routines for putting a number onto the stack, and the one which takes the number off again, with PUSH and POP, and then access them using the CALL instructions. In this the actual machine code instructions, which would otherwise do the job, but which are more complicated to use, are emulated.

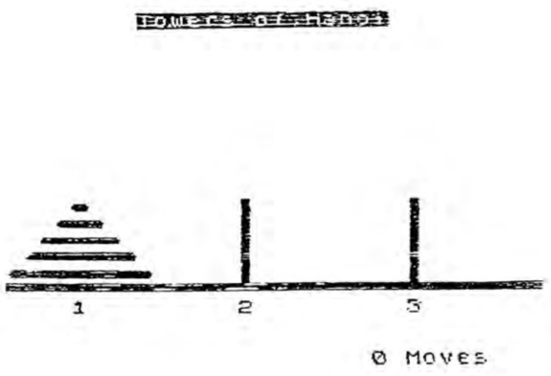
```
org 30000
equ 28000 STK-P
30000 3A 30 75      ld a,(STK-P)
30003 21 30 75      ld hl,STK-P
30006 34            inc (hl)
```

```
30007 21 60 6D      ld hl,28000
30010 7E            ld a,(hl)
30011 23            inc hl
```

Now for an interlude. The BASIC program below is for a game which operates on the same principle as the stack itself. Whilst not written in machine code, the idea of the game is directly related to this chapter, and more directly to the use of the stack. The principle of the game is very old, and may already have been encountered elsewhere. Called The Towers of Hanoi, the idea is to transfer a stack of 5 rings of ascending height from one pin or tower to the final pin or tower. One problem being that they must be in the same ascending order as when they started. Additionally only one ring may be moved each time and must come off the top and no "larger" ring can be placed on top of a "smaller" one. This will give you some idea of the problems which can be encountered with stacks, especially if you lose track of what is on the top!

```
   2 GO SUB 2000
   5 BORDER 6: PAPER 6: INK 0: C
LS
   7 DATA "1     ▬    ","2    ▬▬
 ","3  ▬▬▬▬    ","4 ▬▬▬▬▬▬▬    ","5
▬▬▬▬▬▬▬▬▬▬ ","0    ▌    "
  10 RESTORE 7: LET C=10: LET M=
0: DIM A$(16,11): DIM C$(5,11):
LET W=0: DIM B$(2,1): FOR D=1 TO
 16
  20 IF D<7 THEN READ A$(D): GO
TO 40
  30 LET A$(D)=A$(6)
  40 NEXT D
 400 FOR X=1 TO 5: LET C$(X)=A$(
X): NEXT X
 470 PRINT AT 0,5;"Towers of Han
oi"
 475 PRINT AT 20,22;M;" Moves"
 480 FOR D=0 TO 31: PRINT AT 16,
D;"■": NEXT D: PRINT AT 17,4; IN
K 1;"1";TAB 14;"2";TAB 24;"3"
 500 FOR B=1 TO 15: LET C=C+1: P
RINT AT C,INT (B/5-.1)*10;A$(B,2
 TO )
 510 IF C=15 THEN LET C=10
 520 NEXT B
 530 IF W=1 THEN GO TO 1600
 580 INPUT "FROM PIN?"; LINE B$(
1),"TO PIN?"; LINE B$(2): PRINT
AT 21,15;"                 ": IF B$(
1)<STR$ 1 OR B$(1)>STR$ 3 OR B$(
2)<STR$ 1 OR B$(2)>STR$ 3 THEN G
O TO 1500
 600 FOR Z=1 TO 5
 610 IF A$((VAL B$(1)-1)*5+Z)<>A
$(16) THEN GO TO 640
 615 IF Z=5 AND A$((VAL B$(1)-1)
*5+Z)=A$(16) THEN GO TO 1500
 620 NEXT Z
 640 FOR Y=5 TO 1 STEP -1
 660 IF A$((VAL B$(2)-1)*5+Y)=A$
(16) THEN GO TO 1000
 680 NEXT Y
1000 IF Y=5 THEN GO TO 1010
1003 IF A$((VAL B$(2)-1)*5+Y+1)<
A$((VAL B$(1)-1)*5+Z) THEN GO TO
 1500
1010 LET A$((VAL B$(2)-1)*5+Y)=A
$((VAL B$(1)-1)*5+Z)
1020 LET A$((VAL B$(1)-1)*5+Z)=A
$(16)
1030 LET M=M+1
1040 FOR D=1 TO 5: IF A$(D+10)<>
C$(D) THEN GO TO 470
1050 NEXT D: LET W=1: GO TO 470
1500 BEEP 1,-20: PRINT AT 21,15;
 FLASH 1;"ILLEGAL MOVE"
1520 GO TO 580
1600 PRINT AT 5,11; FLASH 1;"WEL
L DONE"; FLASH 0;''TAB 5;"You di
d it in ";M;" Moves,": IF M=31 T
HEN PRINT TAB 5; FLASH 1;"Which
cannot be beaten!!!": FLASH 0: F
OR x=0 TO 255: OUT 254,x: NEXT x
: GO TO 1605
1601 PRINT TAB 5;"But it can be
bettered."
1610 INPUT "ANOTHER GAME?-y/n";Q
$: IF Q$<>"Y" AND Q$<>"y" THEN P
RINT AT 21,0;"bye...": STOP
1620 RUN
2000 RESTORE 3000
2005 FOR y=0 TO 5
2010 FOR x=0 TO 7
2020 READ chr: POKE 65368+y*8+x,
chr
2030 NEXT x
2040 NEXT y
2050 RETURN
3000 DATA 0,0,BIN 00111111,BIN 0
1111111,BIN 01111111,BIN 0011111
1,0,0
```

```
3010 DATA 0,0,255,255,255,255,0,
0
3020 DATA 0,0,BIN 11111100,BIN 1
1111110,BIN 11111110,BIN 1111110
0,0,0
3030 DATA 60,60,60,60,60,60,60,6
0
3040 DATA 0,24,60,60,60,60,60,60
3050 DATA 0,0,BIN 01111110,255,2
55,BIN 01111110,0,0
```



BASIC, indirectly and directly, uses 3 separate stacks for its operations: The GO SUB stack which holds the return destinations of all the BASIC sub-routines. The Calculator stack which is used for all BASIC numerical operations. Finally the machine stack, which is used by the CPU itself, and is accessible via machine code. To locate or use the machine stack from BASIC is not easy and to go into machine code before use is always advisable.

Earlier we looked at the INC and DEC instructions and how they cause the contents of a register or location to be INCremented or DECremented by one. There are some obvious uses to which they are ideally suited. One such example is the transfer of the contents of one block of memory to another. In this case we use two pointers to indicate these two areas which, for convenience, can be regarded as the Source Block and the Destination Block. The INC and DEC commands are then used to increase or decrease the values held by the pointers, and so avoid them having to be reloaded before each re-execution of the transfer. For example:

```
ld hl,source
ld de,destination
ld a,(hl)
ld (de),a
inc hl

inc de
ld a,(hl)
ld (de),a
inc hl
inc de
"

"
```

This might well be satisfactory for moving one or two bytes, but for anything more ambitious far too laborious. Fortunately the LOOP comes to our rescue, but before using this command we must specify, within a register, the number of 'times' we require the loop to be executed. This is done as follow:

```
ld hl,source
ld de,destination
ld bc,no. of times
loop ld a,(hl)
ld (de),a
inc de
inc hl
dec bc
ld a,b
add a,c
jr nz,loop
ret
```

By adding the contents of the High Byte (B) to those of the Low Byte (C) it is possible to check whether or not the content value of the Register Pair is Zero. This therefore does provide us with a relatively simple means whereby we can transfer a block of memory.

The routine above can be further simplified using two commands that require only two bytes each (INC and DEC require only one byte each), to such an extent that all but four of the commands in the routine can be eliminated. These two new commands are "LDIR" and "LDDR", and their definitions are:

LDIR	=	"Conditional memory-to-memory transfer with auto-increment of memory pointer registers, and auto-decrement of a byte-count register"
LDDR	=	"Conditional memory-to-memory transfer with auto-decrement of memory pointer registers, and auto-decrement of a byte-count register".

All this means that, with the LDIR command we can load the HL register-pair with the start of the source code; the DE register-pair with the start of the destination area of memory, and the BC register-pair with the number of bytes to be transferred. Then by use of the LDIR command the number of bytes specified (as in BC) are moved from location HL to location DE, with the values of HL and DE increasing by one each time. The value in BC will be decremented by one each time until it reaches zero, whereby the command is concluded and the computer moves on to the next command. The LDDR command operates in precisely the same fashion, except of course that the contents of the two pointers are decreased rather than increased. A routine using these commands would look like this:

```
LD HL,source
LD DE,destination
LD BC,bytes
LDIR
RET
```

For example, a routine that moves the whole contents of the screen would appear like this:

```
org 25501
25501 21 00 40      ld hl,16384
25504 11 00 64      ld de,25600
25507 01 00 1C      ld bc,7168
25510 ED B0         ldir
25512 C9            ret
25513 21 00 64      ld hl,25600
25516 11 00 40      ld de,16384
25519 01 00 1C      ld bc,7168
25522 ED B0         ldir
25524 C9            ret
```

Now type LOAD in the BASIC 'draw' program given earlier, store a screen using the first of the two above-routines, clear the screen (CLS), and transfer it back again with the second routine. Note the incredible speed of this command.

This brings us to the end of this journey through the essentials of machine code programming — a journey which I trust has

been both enjoyable and informative. If indeed it has left you with a desire to delve deeper into the subject itself, then it has been worthwhile. If it has left you equipped to utilise the information that is to be found in more advanced texts, then truly I have succeeded.

GOOD PROGRAMMING

J.H.C.W. April 1983.

APPENDIX A

00	0	00000000
01	1	00000001
02	2	00000010
03	3	00000011
04	4	00000100
05	5	00000101
06	6	00000110
07	7	00000111
08	8	00001000
09	9	00001001
0A	10	00001010
0B	11	00001011
0C	12	00001100
0D	13	00001101
0E	14	00001110
0F	15	00001111
10	16	00010000
11	17	00010001
12	18	00010010
13	19	00010011
14	20	00010100
15	21	00010101
16	22	00010110
17	23	00010111
18	24	00011000
19	25	00011001
1A	26	00011010
1B	27	00011011
1C	28	00011100
1D	29	00011101
1E	30	00011110
1F	31	00011111
20	32	00100000
21	33	00100001
22	34	00100010
23	35	00100011
24	36	00100100
25	37	00100101
26	38	00100110
27	39	00100111
28	40	00101000
29	41	00101001
2A	42	00101010

2B	43	00101011
2C	44	00101100
2D	45	00101101
2E	46	00101110
2F	47	00101111
30	48	00110000
31	49	00110001
32	50	00110010
33	51	00110011
34	52	00110100
35	53	00110101
36	54	00110110
37	55	00110111
38	56	00111000
39	57	00111001
3A	58	00111010
3B	59	00111011
3C	60	00111100
3D	61	00111101
3E	62	00111110
3F	63	00111111
40	64	01000000
41	65	01000001
42	66	01000010
43	67	01000011
44	68	01000100
45	69	01000101
46	70	01000110
47	71	01000111
48	72	01001000
49	73	01001001
4A	74	01001010
4B	75	01001011
4C	76	01001100
4D	77	01001101
4E	78	01001110
4F	79	01001111
50	80	01010000
51	81	01010001
52	82	01010010
53	83	01010011
54	84	01010100
55	85	01010101
56	86	01010110
57	87	01010111
58	88	01011000
59	89	01011001
5A	90	01011010
5B	91	01011011
5C	92	01011100
5D	93	01011101
5E	94	01011110
5F	95	01011111
60	96	01100000

61	97	01100001
62	98	01100010
63	99	01100011
64	100	01100100
65	101	01100101
66	102	01100110
67	103	01100111
68	104	01101000
69	105	01101001
6A	106	01101010
6B	107	01101011
6C	108	01101100
6D	109	01101101
6E	110	01101110
6F	111	01101111
70	112	01110000
71	113	01110001
72	114	01110010
73	115	01110011
74	116	01110100
75	117	01110101
76	118	01110110
77	119	01110111
78	120	01111000
79	121	01111001
7A	122	01111010
7B	123	01111011
7C	124	01111100
7D	125	01111101
7E	126	01111110
7F	127	01111111
80	128	10000000
81	129	10000001
82	130	10000010
83	131	10000011
84	132	10000100
85	133	10000101
86	134	10000110
87	135	10000111
88	136	10001000
89	137	10001001
8A	138	10001010
8B	139	10001011
8C	140	10001100
8D	141	10001101
8E	142	10001110
8F	143	10001111
90	144	10010000
91	145	10010001
92	146	10010010
93	147	10010011
94	148	10010100
95	149	10010101
96	150	10010110

97	151	10010111
98	152	10011000
99	153	10011001
9A	154	10011010
9B	155	10011011
9C	156	10011100
9D	157	10011101
9E	158	10011110
9F	159	10011111
A0	160	10100000
A1	161	10100001
A2	162	10100010
A3	163	10100011
A4	164	10100100
A5	165	10100101
A6	166	10100110
A7	167	10100111
A8	168	10101000
A9	169	10101001
AA	170	10101010
AB	171	10101011
AC	172	10101100
AD	173	10101101
AE	174	10101110
AF	175	10101111
B0	176	10110000
B1	177	10110001
B2	178	10110010
B3	179	10110011
B4	180	10110100
B5	181	10110101
B6	182	10110110
B7	183	10110111
B8	184	10111000
B9	185	10111001
BA	186	10111010
BB	187	10111011
BC	188	10111100
BD	189	10111101
BE	190	10111110
BF	191	10111111
C0	192	11000000
C1	193	11000001
C2	194	11000010
C3	195	11000011
C4	196	11000100
C5	197	11000101
C6	198	11000110
C7	199	11000111
C8	200	11001000
C9	201	11001001
CA	202	11001010
CB	203	11001011
CC	204	11001100
CD	205	11001101
CE	206	11001110
CF	207	11001111
D0	208	11010000
D1	209	11010001
D2	210	11010010
D3	211	11010011
D4	212	11010100
D5	213	11010101
D6	214	11010110
D7	215	11010111
D8	216	11011000
D9	217	11011001
DA	218	11011010
DB	219	11011011
DC	220	11011100
DD	221	11011101
DE	222	11011110
DF	223	11011111
E0	224	11100000
E1	225	11100001
E2	226	11100010
E3	227	11100011
E4	228	11100100
E5	229	11100101
E6	230	11100110
E7	231	11100111
E8	232	11101000
E9	233	11101001
EA	234	11101010
EB	235	11101011
EC	236	11101100
ED	237	11101101
EE	238	11101110
EF	239	11101111
F0	240	11110000
F1	241	11110001
F2	242	11110010
F3	243	11110011
F4	244	11110100
F5	245	11110101
F6	246	11110110
F7	247	11110111
F8	248	11111000
F9	249	11111001
FA	250	11111010
FB	251	11111011
FC	252	11111100
FD	253	11111101
FE	254	11111110
FF	255	11111111

```
  10 FOR a=0 TO 255
  20 GO SUB 400: LPRINT "          "
;a;"          ";: GO SUB 200: LPRINT

  30 NEXT a
 200 LET n=a
 210 FOR x=7 TO 0 STEP -1
 220 IF INT (n/(2↑x))=1 THEN GO
TO 300
 230 LPRINT 0;
 240 NEXT x
 250 RETURN
 300 LPRINT 1;
 310 LET n=n-2↑x
 320 GO TO 240
 400 LET n=a
 410 LET x=INT (n/16)
 420 LET y=((n/16)-INT (n/16))*1
6
 430 IF x>9 THEN LET x=x+7
 440 IF y>9 THEN LET y=y+7
 450 LET x=x+48: LET y=y+48
 460 LPRINT CHR$ x;CHR$ y;
 470 RETURN
```

APPENDIX B

Z80 OP CODES   sorted by MNEMONIC

Op code	Hex	Decimal
ADC A,(HL)	8E	142
ADC A,(IX+d)	DD8Edd	221,142,XX
ADC A,(IY+d)	FD8Edd	253,142,XX
ADC A,A	8F	143
ADC A,B	88	136
ADC A,C	89	137
ADC A,D	8A	138
ADC A,E	8B	139
ADC A,H	8C	140
ADC A,L	8D	141
ADC A,xx	CExx	206,XX
ADC HL,BC	ED4A	237,74
ADC HL,DE	ED5A	237,90
ADC HL,HL	ED6A	237,106
ADC HL,SP	ED7A	237,122
ADD A,(HL)	86	134
ADD A,(IX+d)	DD86dd	221,134,XX
ADD A,(IY+d)	FD86dd	253,134,XX
ADD A,A	87	135
ADD A,B	80	128
ADD A,C	81	129
ADD A,D	82	130
ADD A,E	83	131
ADD A,H	84	132
ADD A,L	85	133
ADD A,xx	C6xx	198,XX
ADD HL,BC	09	9
ADD HL,DE	19	25
ADD HL,HL	29	41
ADD HL,SP	39	57
ADD IX,BC	DD09	221,9

Op code	Hex	Decimal
ADD IX,DE	DD19	221,25
ADD IX,IX	DD29	221,41
ADD IX,SP	DD39	221,57
ADD IY,BC	FD09	253,9
ADD IY,DE	FD19	253,25
ADD IY,IY	FD29	253,41
ADD IY,SP	FD39	253,57
AND (HL)	A6	166
AND (IX+d)	DDA6dd	221,166,XX
AND (IY+d)	FDA6dd	253,166,XX
AND A	A7	167
AND B	A0	160
AND C	A1	161
AND D	A2	162
AND E	A3	163
AND H	A4	164
AND L	A5	165
AND xx	E6xx	230,XX
BIT 0,(HL)	CB46	203,70
BIT 0,(IX+d)	DDCBdd46	221,203,XX,70
BIT 0,(IY+d)	FDCBdd46	253,203,XX,70
BIT 0,A	CB47	203,71
BIT 0,B	CB40	203,64
BIT 0,C	CB41	203,65
BIT 0,D	CB42	203,66
BIT 0,E	CB43	203,67
BIT 0,H	CB44	203,68
BIT 0,L	CB45	203,69
BIT 1,(HL)	CB4E	203,78
BIT 1,(IX+d)	DDCBdd4E	221,203,XX,78
BIT 1,(IY+d)	FDCBdd4E	253,203,XX,78
BIT 1,A	CB4F	203,79
BIT 1,B	CB48	203,72
BIT 1,C	CB49	203,73
BIT 1,D	CB4A	203,74
BIT 1,E	CB4B	203,75
BIT 1,H	CB4C	203,76
BIT 1,L	CB4D	203,77
BIT 2,(HL)	CB56	203,86
BIT 2,(IX+d)	DDCBdd56	221,203,XX,86
BIT 2,(IY+d)	FDCBdd56	253,203,XX,86
BIT 2,A	CB57	203,87

Op code	Hex	Decimal
BIT 2,B	CB50	203,80
BIT 2,C	CB51	203,81
BIT 2,D	CB52	203,82
BIT 2,E	CB53	203,83
BIT 2,H	CB54	203,84
BIT 2,L	CB55	203,85
BIT 3,(HL)	CB5E	203,94
BIT 3,(IX+d)	DDCBdd5E	221,203,XX,94
BIT 3,(IY+d)	FDCBdd5E	253,203,XX,94
BIT 3,A	CB5F	203,95
BIT 3,B	CB58	203,88
BIT 3,C	CB59	203,89
BIT 3,D	CB5A	203,90
BIT 3,E	CB5B	203,91
BIT 3,H	CB5C	203,92
BIT 3,L	CB5D	203,93
BIT 4,(HL)	CB66	203,102
BIT 4,(IX+d)	DDCBdd66	221,203,XX,102
BIT 4,(IY+d)	FDCBdd66	253,203,XX,102
BIT 4,A	CB67	203,103
BIT 4,B	CB60	203,96
BIT 4,C	CB61	203,97
BIT 4,D	CB62	203,98
BIT 4,E	CB63	203,99
BIT 4,H	CB64	203,100
BIT 4,L	CB65	203,101
BIT 5,(HL)	CB6E	203,110
BIT 5,(IX+d)	DDCBdd6E	221,203,XX,110
BIT 5,(IY+d)	FDCBdd6E	253,203,XX,110
BIT 5,A	CB6F	203,111
BIT 5,B	CB68	203,104
BIT 5,C	CB69	203,105
BIT 5,D	CB6A	203,106
BIT 5,E	CB6B	203,107
BIT 5,H	CB6C	203,108
BIT 5,L	CB6D	203,109
BIT 6,(HL)	CB76	203,118
BIT 6,(IX+d)	DDCBdd76	221,203,XX,118
BIT 6,(IY+d)	FDCBdd76	253,203,XX,118
BIT 6,A	CB77	203,119
BIT 6,B	CB70	203,112
BIT 6,C	CB71	203,113

Op code	Hex	Decimal
BIT 6,D	CB72	203,114
BIT 6,E	CB73	203,115
BIT 6,H	CB74	203,116
BIT 6,L	CB75	203,117
BIT 7,(HL)	CB7E	203,126
BIT 7,(IX+d)	DDCBdd7E	221,203,XX,126
BIT 7,(IY+d)	FDCBdd7E	253,203,XX,126
BIT 7,A	CB7F	203,127
BIT 7,B	CB78	203,120
BIT 7,C	CB79	203,121
BIT 7,D	CB7A	203,122
BIT 7,E	CB7B	203,123
BIT 7,H	CB7C	203,124
BIT 7,L	CB7D	203,125
CALL C,xxxx	DCxxxx	220,XX,XX
CALL M,xxxx	FCxxxx	252,XX,XX
CALL NC,xxxx	D4xxxx	212,XX,XX
CALL NZ,xxxx	C4xxxx	196,XX,XX
CALL P,xxxx	F4xxxx	244,XX,XX
CALL PE,xxxx	ECxxxx	236,XX,XX
CALL PO,xxxx	E4xxxx	228,XX,XX
CALL xxxx	CDxxxx	205,XX,XX
CALL Z,xxxx	CCxxxx	204,XX,XX
CCF	3F	63
CP (HL)	BE	190
CP (IX+d)	DDBEdd	221,190,XX
CP (IY+d)	FDBEdd	253,190,XX
CP A	BF	191
CP B	B8	184
CP C	B9	185
CP D	BA	186
CP E	BB	187
CP H	BC	188
CP L	BD	189
CP xx	FExx	254,XX
CPD	EDA9	237,169
CPDR	EDB9	237,185
CPI	EDA1	237,161
CPIR	EDB1	237,177
CPL	2F	47
DAA	27	39
DEC (HL)	35	53

Op code	Hex	Decimal
DEC (IX+d)	DD35dd	221,53,XX
DEC (IY+d)	FD35dd	253,53,XX
DEC A	3D	61
DEC B	05	5
DEC BC	0B	11
DEC C	0D	13
DEC D	15	21
DEC DE	1B	27
DEC E	1D	29
DEC H	25	37
DEC HL	2B	43
DEC IX	DD2B	221,43
DEC IY	FD2B	253,43
DEC L	2D	45
DEC SP	3B	59
DI	F3	243
DJNZ xx	10xx	16,XX
EI	FB	251
EX (SP),HL	E3	227
EX (SP),IX	DDE3	221,227
EX (SP),IY	FDE3	253,227
EX AF,AF	08	8
EX DE,HL	EB	235
Exx	D9	217
HALT	76	118
IM 0	ED46	237,70
IM 1	ED56	237,86
IM 2	ED5E	237,94
IN A,(C)	ED78	237,120
IN A,(xx)	DBxx	219,XX
IN B,(C)	ED40	237,64
IN C,(C)	ED48	237,72
IN D,(C)	ED50	237,80
IN E,(C)	ED58	237,88
IN H,(C)	ED60	237,96
IN L,(C)	ED68	237,104
INC (HL)	34	52
INC (IX+d)	DD34dd	221,52,XX
INC (IY+d)	FD34dd	253,52,XX
INC A	3C	60
INC B	04	4
INC BC	03	3

Op code	Hex	Decimal
INC C	0C	12
INC D	14	20
INC DE	13	19
INC E	1C	28
INC H	24	36
INC HL	23	35
INC IX	DD23	221,35
INC IY	FD23	253,35
INC L	2C	44
INC SP	33	51
IND	EDAA	237,170
INDR	EDBA	237,186
INI	EDA2	237,162
INIR	EDB2	237,178
JP (HL)	E9	233
JP (IX)	DDE9	221,233
JP (IY)	FDE9	253,233
JP C,xxxx	DAxxxx	218,XX,XX
JP M,xxxx	FAxxxx	250,XX,XX
JP NC,xxxx	D2xxxx	210,XX,XX
JP NZ,xxxx	C2xxxx	194,XX,XX
JP P,xxxx	F2xxxx	242,XX,XX
JP PE,xxxx	EAxxxx	234,XX,XX
JP PO,xxxx	E2xxxx	226,XX,XX
JP xxxx	C3xxxx	195,XX,XX
JP Z,xxxx	CAxxxx	202,XX,XX
JR C,xx	38xx	56,XX
JR NC,xx	30xx	48,XX
JR NZ,xx	20xx	32,XX
JR xx	18xx	24,XX
JR Z,xx	28xx	40,XX
LD (BC),A	02	2
LD (DE),A	12	18
LD HL,(xxxx)	2Axxxx	42,XX,XX
LD (HL),A	77	119
LD (HL),B	70	112
LD (HL),C	71	113
LD (HL),D	72	114
LD (HL),E	73	115
LD (HL),H	74	116
LD (HL),L	75	117
LD (HL),xx	36xx	54,XX

Op code	Hex	Decimal
LD (IX+d),A	DD77dd	221,119,XX
LD (IX+d),B	DD70dd	221,112,XX
LD (IX+d),C	DD71dd	221,113,XX
LD (IX+d),D	DD72dd	221,114,XX
LD (IX+d),E	DD73dd	221,115,XX
LD (IX+d),H	DD74dd	221,116,XX
LD (IX+d),L	DD75dd	221,117,XX
LD (IX+d),xx	DD36ddxx	221,54,XX,XX
LD (IY+d),A	FD77dd	253,119,XX
LD (IY+d),B	FD70dd	253,112,XX
LD (IY+d),C	FD71dd	253,113,XX
LD (IY+d),D	FD72dd	253,114,XX
LD (IY+d),E	FD73dd	253,115,XX
LD (IY+d),H	FD74dd	253,116,XX
LD (IY+d),L	FD75dd	253,117,XX
LD (IY+d),xx	FD36ddxx	253,54,XX,XX
LD (xxxx),A	32xxxx	50,XX,XX
LD (xxxx),BC	ED43xxxx	237,67,XX,XX
LD (xxxx),DE	ED53xxxx	237,83,XX,XX
LD (xxxx),HL	22xxxx	34,XX,XX
LD (xxxx),IX	DD22xxxx	221,34,XX,XX
LD (xxxx),IY	FD22xxxx	253,34,XX,XX
LD (xxxx),SP	ED73xxxx	237,115,XX,XX
LD A,(BC)	0A	10
LD A,(DE)	1A	26
LD A,(HL)	7E	126
LD A,(IX+d)	DD7Edd	221,126,XX
LD A,(IY+d)	FD7Edd	253,126,XX
LD A,(xxxx)	3Axxxx	58,XX,XX
LD A,A	7F	127
LD A,B	78	120
LD A,C	79	121
LD A,D	7A	122
LD A,E	7B	123
LD A,H	7C	124
LD A,I	ED57	237,87
LD A,L	7D	125
LD A,R	ED5F	237,95
LD A,xx	3Exx	62,XX
LD B,(HL)	46	70
LD B,(IX+d)	DD46dd	221,70,XX
LD B,(IY+d)	FD46dd	253,70,XX

Op code		Hex	Decimal
LD	B,A	47	71
LD	B,B	40	64
LD	B,C	41	65
LD	B,D	42	66
LD	B,E	43	67
LD	B,H	44	68
LD	B,L	45	69
LD	B,xx	06xx	6,XX
LD	BC,(xxxx)	ED4Bxxxx	237,75,XX,XX
LD	BC,xxxx	01xxxx	1,XX,XX
LD	C,(HL)	4E	78
LD	C,(IX+d)	DD4Edd	221,78,XX
LD	C,(IY+d)	FD4Edd	253,78,XX
LD	C,A	4F	79
LD	C,B	48	72
LD	C,C	49	73
LD	C,D	4A	74
LD	C,E	4B	75
LD	C,H	4C	76
LD	C,L	4D	77
LD	C,xx	0Exx	14,XX
LD	D,(HL)	56	86
LD	D,(IX+d)	DD56dd	221,86,XX
LD	D,(IY+d)	FD56dd	253,86,XX
LD	D,A	57	87
LD	D,B	50	80
LD	D,C	51	81
LD	D,D	52	82
LD	D,E	53	83
LD	D,H	54	84
LD	D,L	55	85
LD	D,xx	16xx	22,XX
LD	DE,(xxxx)	ED5Bxxxx	237,91,XX,XX
LD	DE,xxxx	11xxxx	17,XX,XX
LD	E,(HL)	5E	94
LD	E,(IX+d)	DD5Edd	221,94,XX
LD	E,(IY+d)	FD5Edd	253,94,XX
LD	E,A	5F	95
LD	E,B	58	88
LD	E,C	59	89
LD	E,D	5A	90
LD	E,E	5B	91

Op code		Hex	Decimal
LD	E,H	5C	92
LD	E,L	5D	93
LD	E,xx	1Exx	30,XX
LD	H,(HL)	66	102
LD	H,(IX+d)	DD66dd	221,102,XX
LD	H,(IY+d)	FD66dd	253,102,XX
LD	H,A	67	103
LD	H,B	60	96
LD	H,C	61	97
LD	H,D	62	98
LD	H,E	63	99
LD	H,H	64	100
LD	H,L	65	101
LD	H,xx	26xx	38,XX
LD	HL,xxxx	21xxxx	33,XX,XX
LD	I,A	ED47	237,71
LD	IX,(xxxx)	DD2Axxxx	221,42,XX,XX
LD	IX,xxxx	DD21xxxx	221,33,XX,XX
LD	IY,(xxxx)	FD2Axxxx	253,42,XX,XX
LD	IY,xxxx	FD21xxxx	253,33,XX,XX
LD	L,(HL)	6E	110
LD	L,(IX+d)	DD6Edd	221,110,XX
LD	L,(IY+d)	FD6Edd	253,110,XX
LD	L,A	6F	111
LD	L,B	68	104
LD	L,C	69	105
LD	L,D	6A	106
LD	L,E	6B	107
LD	L,H	6C	108
LD	L,L	6D	109
LD	L,xx	2Exx	46,XX
LD	R,A	ED4F	237,79
LD	SP,(xxxx)	ED7Bxxxx	237,123,XX,XX
LD	SP,HL	F9	249
LD	SP,IX	DDF9	221,249
LD	SP,IY	FDF9	253,249
LD	SP,xxxx	31xxxx	49,XX,XX
LDD		EDA8	237,168
LDDR		EDB8	237,184
LDI		EDA0	237,160
LDIR		EDB0	237,176
NEG		ED44	237,68

Op code	Hex	Decimal
NOP	00	0
OR (HL)	B6	182
OR (IX+d)	DDB6dd	221,182,XX
OR (IY+d)	FDB6dd	253,182,XX
OR A	B7	183
OR B	B0	176
OR C	B1	177
OR D	B2	178
OR E	B3	179
OR H	B4	180
OR L	B5	181
OR xx	F6xx	246,XX
OTDR	EDBB	237,187
OTIR	EDB3	237,179
OUT (C),A	ED79	237,121
OUT (C),B	ED41	237,65
OUT (C),C	ED49	237,73
OUT (C),D	ED51	237,81
OUT (C),E	ED59	237,89
OUT (C),H	ED61	237,97
OUT (C),L	ED69	237,105
OUT (xx),A	D3xx	211,XX
OUTD	EDAB	237,171
OUTI	EDA3	237,163
POP AF	F1	241
POP BC	C1	193
POP DE	D1	209
POP HL	E1	225
POP IX	DDE1	221,225
POP IY	FDE1	253,225
PUSH AF	F5	245
PUSH BC	C5	197
PUSH DE	D5	213
PUSH HL	E5	229
PUSH IX	DDE5	221,229
PUSH IY	FDE5	253,229
RES 0,(HL)	CB86	203,134
RES 0,(IX+d)	DDCBdd86	221,203,XX,134
RES 0,(IY+d)	FDCBdd86	253,203,XX,134
RES 0,A	CB87	203,135
RES 0,B	CB80	203,128
RES 0,C	CB81	203,129

Op code	Hex	Decimal
RES 0,D	CB82	203,130
RES 0,E	CB83	203,131
RES 0,H	CB84	203,132
RES 0,L	CB85	203,133
RES 1,(HL)	CB8E	203,142
RES 1,(IX+d)	DDCBdd8E	221,203,XX,142
RES 1,(IY+d)	FDCBdd8E	253,203,XX,142
RES 1,A	CB8F	203,143
RES 1,B	CB88	203,136
RES 1,C	CB89	203,137
RES 1,D	CB8A	203,138
RES 1,E	CB8B	203,139
RES 1,H	CB8C	203,140
RES 1,L	CB8D	203,141
RES 2,(HL)	CB96	203,150
RES 2,(IX+d)	DDCBdd96	221,203,XX,150
RES 2,(IY+d)	FDCBdd96	253,203,XX,150
RES 2,A	CB97	203,151
RES 2,B	CB90	203,144
RES 2,C	CB91	203,145
RES 2,D	CB92	203,146
RES 2,E	CB93	203,147
RES 2,H	CB94	203,148
RES 2,L	CB95	203,149
RES 3,(HL)	CB9E	203,158
RES 3,(IX+d)	DDCBdd9E	221,203,XX,158
RES 3,(IY+d)	FDCBdd9E	253,203,XX,158
RES 3,A	CB9F	203,159
RES 3,B	CB98	203,152
RES 3,C	CB99	203,153
RES 3,D	CB9A	203,154
RES 3,E	CB9B	203,155
RES 3,H	CB9C	203,156
RES 3,L	CB9D	203,157
RES 4,(HL)	CBA6	203,166
RES 4,(IX+d)	DDCBddA6	221,203,XX,166
RES 4,(IY+d)	FDCBddA6	253,203,XX,166
RES 4,A	CBA7	203,167
RES 4,B	CBA0	203,160
RES 4,C	CBA1	203,161
RES 4,D	CBA2	203,162
RES 4,E	CBA3	203,163

Op code		Hex	Decimal
RES	4,H	CBA4	203,164
RES	4,L	CBA5	203,165
RES	5,(HL)	CBAE	203,174
RES	5,(IX+d)	DDCBddAE	221,203,XX,174
RES	5,(IY+d)	FDCBddAE	253,203,XX,174
RES	5,A	CBAF	203,175
RES	5,B	CBA8	203,168
RES	5,C	CBA9	203,169
RES	5,D	CBAA	203,170
RES	5,E	CBAB	203,171
RES	5,H	CBAC	203,172
RES	5,L	CBAD	203,173
RES	6,(HL)	CBB6	203,182
RES	6,(IX+d)	DDCBddB6	221,203,XX,182
RES	6,(IY+d)	FDCBddB6	253,203,XX,182
RES	6,A	CBB7	203,183
RES	6,B	CBB0	203,176
RES	6,C	CBB1	203,177
RES	6,D	CBB2	203,178
RES	6,E	CBB3	203,179
RES	6,H	CBB4	203,180
RES	6,L	CBB5	203,181
RES	7,(HL)	CBBE	203,190
RES	7,(IX+d)	DDCBddBE	221,203,XX,190
RES	7,(IY+d)	FDCBddBE	253,203,XX,190
RES	7,A	CBBF	203,191
RES	7,B	CBB8	203,184
RES	7,C	CBB9	203,185
RES	7,D	CBBA	203,186
RES	7,E	CBBB	203,187
RES	7,H	CBBC	203,188
RES	7,L	CBBD	203,189
RET		C9	201
RET	C	D8	216
RET	M	F8	248
RET	NC	D0	208
RET	NZ	C0	192
RET	P	F0	240
RET	PE	E8	232
RET	PO	E0	224
RET	Z	C8	200
RETI		ED4D	237,77

Op code		Hex	Decimal
RETN		ED45	237,69
RL	(HL)	CB16	203,22
RL	(IX+d)	DDCBdd16	221,203,XX,22
RL	(IY+d)	FDCBdd16	253,203,XX,22
RL	A	CB17	203,23
RL	B	CB10	203,16
RL	C	CB11	203,17
RL	D	CB12	203,18
RL	E	CB13	203,19
RL	H	CB14	203,20
RL	L	CB15	203,21
RLA		17	23
RLC	(HL)	CB06	203,6
RLC	(IX+d)	DDCBdd06	221,203,XX,6
RLC	(IY+d)	FDCBdd06	253,203,XX,6
RLC	A	CB07	203,7
RLC	B	CB00	203,0
RLC	C	CB01	203,1
RLC	D	CB02	203,2
RLC	E	CB03	203,3
RLC	H	CB04	203,4
RLC	L	CB05	203,5
RLCA		07	7
RLD		ED6F	237,111
RR	(HL)	CB1E	203,30
RR	(IX+d)	DDCBdd1E	221,203,XX,30
RR	(IY+d)	FDCBdd1E	253,203,XX,30
RR	A	CB1F	203,31
RR	B	CB18	203,24
RR	C	CB19	203,25
RR	D	CB1A	203,26
RR	E	CB1B	203,27
RR	H	CB1C	203,28
RR	L	CB1D	203,29
RRA		1F	31
RRC	(HL)	CB0E	203,14
RRC	(IX+d)	DDCBdd0E	221,203,XX,14
RRC	(IY+d)	FDCBdd0E	253,203,XX,14
RRC	A	CB0F	203,15
RRC	B	CB08	203,8
RRC	C	CB09	203,9
RRC	D	CB0A	203,10

Op code	Hex	Decimal
RRC E	CB0B	203,11
RRC H	CB0C	203,12
RRC L	CB0D	203,13
RRCA	0F	15
RRD	ED67	237,103
RST 0	C7	199
RST 10h	D7	215
RST 18h	DF	223
RST 20h	E7	231
RST 28h	EF	239
RST 30h	F7	247
RST 38h	FF	255
RST 8	CF	207
SBC A,(HL)	9E	158
SBC A,(IX+d)	DD9Edd	221,158,XX
SBC A,(IY+d)	FD9Edd	253,158,XX
SBC A,A	9F	159
SBC A,B	98	152
SBC A,C	99	153
SBC A,D	9A	154
SBC A,E	9B	155
SBC A,H	9C	156
SBC A,L	9D	157
SBC A,xx	DExx	222,XX
SBC HL,BC	ED42	237,66
SBC HL,DE	ED52	237,82
SBC HL,HL	ED62	237,98
SBC HL,SP	ED72	237,114
SCF	37	55
SET 0,(HL)	CBC6	203,198
SET 0,(IX+d)	DDCBddC6	221,203,XX,198
SET 0,(IY+d)	FDCBddC6	253,203,XX,198
SET 0,A	CBC7	203,199
SET 0,B	CBC0	203,192
SET 0,C	CBC1	203,193
SET 0,D	CBC2	203,194
SET 0,E	CBC3	203,195
SET 0,H	CBC4	203,196
SET 0,L	CBC5	203,197
SET 1,(HL)	CBCE	203,206
SET 1,(IX+d)	DDCBddCE	221,203,XX,206
SET 1,(IY+d)	FDCBddCE	253,203,XX,206

Op code	Hex	Decimal
SET 1,A	CBCF	203,207
SET 1,B	CBC8	203,200
SET 1,C	CBC9	203,201
SET 1,D	CBCA	203,202
SET 1,E	CBCB	203,203
SET 1,H	CBCC	203,204
SET 1,L	CBCD	203,205
SET 2,(HL)	CBD6	203,214
SET 2,(IX+d)	DDCBddD6	221,203,XX,214
SET 2,(IY+d)	FDCBddD6	253,203,XX,214
SET 2,A	CBD7	203,215
SET 2,B	CBD0	203,208
SET 2,C	CBD1	203,209
SET 2,D	CBD2	203,210
SET 2,E	CBD3	203,211
SET 2,H	CBD4	203,212
SET 2,L	CBD5	203,213
SET 3,(HL)	CBDE	203,222
SET 3,(IX+d)	DDCBddDE	221,203,XX,222
SET 3,(IY+d)	FDCBddDE	253,203,XX,222
SET 3,A	CBDF	203,223
SET 3,B	CBD8	203,216
SET 3,C	CBD9	203,217
SET 3,D	CBDA	203,218
SET 3,E	CBDB	203,219
SET 3,H	CBDC	203,220
SET 3,L	CBDD	203,221
SET 4,(HL)	CBE6	203,230
SET 4,(IX+d)	DDCBddE6	221,203,XX,230
SET 4,(IY+d)	FDCBddE6	253,203,XX,230
SET 4,A	CBE7	203,231
SET 4,B	CBE0	203,224
SET 4,C	CBE1	203,225
SET 4,D	CBE2	203,226
SET 4,E	CBE3	203,227
SET 4,H	CBE4	203,228
SET 4,L	CBE5	203,229
SET 5,(HL)	CBEE	203,238
SET 5,(IX+d)	DDCBddEE	221,203,XX,238
SET 5,(IY+d)	FDCBddEE	253,203,XX,238
SET 5,A	CBEF	203,239
SET 5,B	CBE8	203,232

Op code	Hex	Decimal
SET 5,C	CBE9	203,233
SET 5,D	CBEA	203,234
SET 5,E	CBEB	203,235
SET 5,H	CBEC	203,236
SET 5,L	CBED	203,237
SET 6,(HL)	CBF6	203,246
SET 6,(IX+d)	DDCBddF6	221,203,XX,246
SET 6,(IY+d)	FDCBddF6	253,203,XX,246
SET 6,A	CBF7	203,247
SET 6,B	CBF0	203,240
SET 6,C	CBF1	203,241
SET 6,D	CBF2	203,242
SET 6,E	CBF3	203,243
SET 6,H	CBF4	203,244
SET 6,L	CBF5	203,245
SET 7,(HL)	CBFE	203,254
SET 7,(IX+d)	DDCBffFE	221,203,XX,254
SET 7,(IY+d)	FDCBddFE	253,203,XX,254
SET 7,A	CBFF	203,255
SET 7,B	CBF8	203,248
SET 7,C	CBF9	203,249
SET 7,D	CBFA	203,250
SET 7,E	CBFB	203,251
SET 7,H	CBFC	203,252
SET 7,L	CBFD	203,253
SLA (HL)	CB26	203,38
SLA (IX+d)	DDCBdd26	221,203,XX,38
SLA (IY+d)	FDCBdd26	253,203,XX,38
SLA A	CB27	203,39
SLA B	CB20	203,32
SLA C	CB21	203,33
SLA D	CB22	203,34
SLA E	CB23	203,35
SLA H	CB24	203,36
SLA L	CB25	203,37
SRA (HL)	CB2E	203,46
SRA (IX+d)	DDCBdd2E	221,203,XX,46
SRA (IY+d)	FDCBdd2E	253,203,XX,46
SRA A	CB2F	203,47
SRA B	CB28	203,40
SRA C	CB29	203,41
SRA D	CB2A	203,42

Op code	Hex	Decimal
SRA E	CB2B	203,43
SRA H	CB2C	203,44
SRA L	CB2D	203,45
SRL (HL)	CB3E	203,62
SRL (IX+d)	DDCBdd3E	221,203,XX,62
SRL (IY+d)	FDCBdd3E	253,203,XX,62
SRL A	CB3F	203,63
SRL B	CB38	203,56
SRL C	CB39	203,57
SRL D	CB3A	203,58
SRL E	CB3B	203,59
SRL H	CB3C	203,60
SRL L	CB3D	203,61
SUB (HL)	96	150
SUB (IX+d)	DD96dd	221,150,XX
SUB (IY+d)	FD96dd	253,150,XX
SUB A	97	151
SUB B	90	144
SUB C	91	145
SUB D	92	146
SUB E	93	147
SUB H	94	148
SUB L	95	149
SUB xx	D6xx	214,XX
XOR (HL)	AE	174
XOR (IX+d)	DDAEdd	221,174,XX
XOR (IY+d)	FDAEdd	253,174,XX
XOR A	AF	175
XOR B	A8	168
XOR C	A9	169
XOR D	AA	170
XOR E	AB	171
XOR H	AC	172
XOR L	AD	173
XOR xx	EExx	238,XX

APPENDIX C
LIST OF SUPPLIERS

ACS Software, Dept., MC,
7 Lidgett Crescent,
Roundhay,
Leeds LS8 1HN.

Artic Computing,
396 James Reckitt Avenue,
Hull,
HUB 0JA.

Crystal Computing,
2 Ashton Way,
East Herrington,
Sunderland SR3 3RX.

Picturesque,
6 Corkscrew Hill,
West Wickham,
Kent.

APPENDIX D

Machine Code Instructions

INSTRUCTIONS		FLAGS							
Op code	*	S	Z	–	H	–	P	N	C
ADC A,r		@	@	–	@	–	@	0	@
ADC HL,s		@	@	–	@	–	@	0	@
ADD A,r		@	@	–	@	–	@	0	@
ADD HL,s		–	–	–	@	–	–	0	@
ADD IX,s		–	–	–	@	–	–	0	@
ADD IY,s		–	–	–	@	–	–	0	@
AND r		@	@	–	1	–	@	0	0
BIT b,r		?	@	–	1	–	@	0	0
CALL pq		–	–	–	–	–	–	–	–
CALL c,pq		–	–	–	–	–	–	–	–
CCF		–	–	–	x	–	–	0	@
(the H flag becomes the previous value of the C flag)									
CP r		@	@	–	@	–	@	1	@
CPI		@	x	–	@	–	x	1	–
CPD		@	x	–	@	–	x	1	–
CPIR		@	x	–	@	–	x	1	–
CPDR		@	x	–	@	–	x	1	–
(Z becomes 1 if BC becomes zero. P/V becomes 1 if A = (HL-1))									
CPL		–	–	–	1	–	–	1	–
DAA		@	@	–	@	–	@	–	@
DEC r		@	@	–	@	–	@	1	–
DEC s		–	–	–	–	–	–	–	–
DI		–	–	–	–	–	–	–	–
DJNZ e		–	–	–	–	–	–	–	–
EI		–	–	–	–	–	–	–	–
EX AF, AF'		–	–	–	–	–	–	–	–
EX DE,HL		–	–	–	–	–	–	–	–
EX (SP),HL		–	–	–	–	–	–	–	–
EX (SP),IX		–	–	–	–	–	–	–	–
EX (SP),IY		–	–	–	–	–	–	–	–
EXX		–	–	–	–	–	–	–	–
HALT		–	–	–	–	–	–	–	–
IM 0		–	–	–	–	–	–	–	–
IM 1		–	–	–	–	–	–	–	–
IM 2		–	–	–	–	–	–	–	–
INC r		@	@	–	@	–	@	0	–
INC s		–	–	–	–	–	–	–	–
IN A,(n)		–	–	–	–	–	–	–	–

INSTRUCTIONS	FLAGS							
Op-code	S	Z	–	H	–	P	N	C
IN r,(C)	@	@	–	@	–	@	0	–
INI	?	x	–	?	–	?	1	–
IND	?	x	–	?	–	?	1	–
(Z becomes 1 if B becomes zero)								
INIR	?	1	–	?	–	?	1	–
INDR	?	1	–	?	–	?	1	–
JP pq	–	–	–	–	–	–	–	–
JP c,pq	–	–	–	–	–	–	–	–
JP (HL)	–	–	–	–	–	–	–	–
JP (IX)	–	–	–	–	–	–	–	–
JP (IY)	–	–	–	–	–	–	–	–
JR e	–	–	–	–	–	–	–	–
JR c,e	–	–	–	–	–	–	–	–
LD(BC),A	–	–	–	–	–	–	–	–
LD A,(BC)	–	–	–	–	–	–	–	–
LD (DE),A	–	–	–	–	–	–	–	–
LD A,(DE)	–	–	–	–	–	–	–	–
LD I,A	–	–	–	–	–	–	–	–
LD R,A	–	–	–	–	–	–	–	–
LD A,I	@	@	–	0	–	x	0	–
LD A,R	@	@	–	0	x	0	–	
(P/V is set to interrupt storage flag)								
LD SP,HL	–	–	–	–	–	–	–	–
LD SP,IX	–	–	–	–	–	–	–	–
LD SP,IY	–	–	–	–	–	–	–	–
LD r,r	–	–	–	–	–	–	–	–
LD s,mn	–	–	–	–	–	–	–	–
LD A,(pq)	–	–	–	–	–	–	–	–
LD s,(pq)	–	–	–	–	–	–	–	–
LD (pq),A	–	–	–	–	–	–	–	–
LD (pq),s	–	–	–	–	–	–	–	–
LDI	–	–	–	0	–	x	0	–
LDD	–	–	–	0	–	x	0	–
(P/V becomes 0 if BC becomes 0)								
LDIR	–	–	–	0	–	0	0	–
LDDR	–	–	–	0	–	0	0	–
NEG	@	@	–	@	–	@	1	@
NOP	–	–	–	–	–	–	–	–
OR r	@	@	–	0	–	@	0	0
OUT (n),A	–	–	–	–	–	–	–	–
OUT (C),r	–	–	–	–	–	–	–	–
OUTI	?	x	–	?	–	?	1	–
OUTD	?	x	–	?	–	?	1	–
(Z becomes 1 if B becomes zero)								
OTIR	?	1	–	?	–	?	1	–
OTDR	?	1	–	?	–	?	1	–
POP AF	x	x	x	x	x	x	x	x
(Flags are determined by the byte at the top of the stack)								
POP s	–	–	–	–	–	–	–	–
PUSH AF	–	–	–	–	–	–	–	–
PUSH s	–	–	–	–	–	–	–	–
RES b,r	–	–	–	–	–	–	–	–
RET	–	–	–	–	–	–	–	–
RET c	–	–	–	–	–	–	–	–
RETN	–	–	–	–	–	–	–	–
RETI	–	–	–	–	–	–	–	–
RLA	–	–	–	0	–	–	0	@

INSTRUCTIONS	FLAGS							
Op-code	S	Z	–	H	–	P	N	C
RL r	@	@	–	0	–	@	0	@
RLCA	–	–	–	–	–	–	–	–
RES b,r	–	–	–	–	–	–	–	–
RET	–	–	–	–	–	–	–	–
RET c	–	–	–	–	–	–	–	–
RETN	–	–	–	–	–	–	–	–
RETI	–	–	–	–	–	–	–	–
RLCA	–	–	–	0	–	–	0	@
RRCA	–	–	–	0	–	–	0	@
RLA	–	–	–	0	–	–	0	@
RRA	–	–	–	0	–	–	0	@
RLC r	@	@	–	0	–	@	0	@
RRC r	@	@	–	0	–	@	0	@
RL r	@	@	–	0	–	@	0	@
RR r	@	@	–	0	–	@	0	@
RRD	@	@	–	0	–	@	0	–
RLD	@	@	–	0	–	@	0	–
RST 00	–	–	–	–	–	–	–	–
RST 08	–	–	–	–	–	–	–	–
RST 10	–	–	–	–	–	–	–	–
RST 18	–	–	–	–	–	–	–	–
RST 20	–	–	–	–	–	–	–	–
RST 28	–	–	–	–	–	–	–	–
RST 30	–	–	–	–	–	–	–	–
RST 38	–	–	–	–	–	–	–	–
SBC A,r	@	@	–	@	–	@	1	@
SBC HL,s	@	@	–	@	–	@	1	@
SCF	–	–	–	0	–	–	0	1
SET b,r	–	–	–	–	–	–	–	–
SLA r	@	@	–	0	–	@	0	@
SRA r	@	@	–	0	–	@	0	@
SLR r	@	@	–	0	–	@	0	@
SUB r	@	@	–	@	–	@	1	@
XOR r	@	@	–	0	–	@	0	0

NOTES

NOTES

NOTES