ARGUS BOOKS

ASSEMBLY LANGUAGE PROGRAMMING FOR THE AMSTRAD CPC 464, 664 & 6128

A.P. & D.J. STEPHENSON

Assembly Language Programming for the AMSTRAD CPC 464/664/6128

Assembly Language Programming for the AMSTRAD CPC 464/664/6128

A. P. & D. J. STEPHENSON

ARGUS BOOKS

Argus Books Ltd
1 Golden Square
London W1R 3AB

© Argus Books Limited 1986

ISBN 0 85242 861 8

All rights reserved. No part of this publication
may be reproduced in any form, by print, photography,
microfilm or any other means without written
permission from the publisher

Phototypesetting by Photocomp Limited, Birmingham
Printed and bound by Whitstable Litho

Preface

We hope that this book will persuade readers that machine code, entered with the help of an assembler, is worth learning. The reader should have some experience of BASIC, together with an understanding of simple computer terms. Anyone entirely new to computing and who may have just bought an Amstrad should study the User Instructions first, before tackling this book.

Machine code programs execute in a flash and take up far less memory than an equivalent program written in BASIC. This is why most commercial software, including arcade games, are written in machine code.

Although written for the Amstrad, the book should also be helpful to those who own other machines that incorporate the Z80 microprocessor – the Spectrum for example. Experience of machine code, using 8-bit micros, is never wasted because most of the techniques involved will still apply to the new breed of 16 or 32 bit micros.

Some readers may be a little surprised by the scarcity of 'remarks' in the program listings. For experienced programmers, there can be no denying their importance. For those trying to learn machine code from scratch, the sparse language and cryptic abbreviations used in the typical 'remark' column can be more confusing than helpful. Instead, we have included a line-by-line description in the accompanying text.

We have stressed, quite early in the book, that an assembler is a powerful weapon in the battle against machine code. We realise, however, that some readers may be content to struggle along without one so a simple hex loading program is featured which should reduce the boredom associated with POKEing program bytes into memory.

The complete Z80 instruction set is presented as an Appendix and is laid out in a special format which, we hope, will provide an easy-to-read reference source. Detailed descriptions of the most commonly used Z80 instructions are given in Chapter 6.

Where applicable, descriptions are supported by example program listings. Some of these listings will be found to have

immediate practical value while others may serve as useful guidelines for the development of other programs. Special attention has been given in the early chapters to the design of loops and conditional branching because these are areas which provide a rich breeding ground for bugs. Several full length subroutine listings appear in the later chapters, each supported by detailed explanations which should help those wishing to introduce modifications.

Resident System Extensions, known as RSXs, provide a means of adding extra commands to the operating system repertoire. We have devoted considerable space to the techniques involved, including practical machine code listings which can be called as an RSX from BASIC.

Attention has been drawn to the advantages of writing 'relocatable' machine code so we have shown, with practical examples, how to convert normal code to a relocatable form.

The final chapter explains how the movement of 'shapes' on the screen can be made free from flicker and without the jerkiness which is so irritating in animated sequences.

AP and DJ Stephenson

Contents

1 **Why machine code?** 1

Good points of BASIC. Defects of BASIC. Translating high-level languages. Compilers and interpreters. Assemblers. Binary machine code. Hexadecimal machine code. Mixing BASIC with machine code.

2 **Machine hardware** 10

Hardware. Logic chips. The Z80 microprocessor. The address bus. The data bus. The control bus. The clock. RAM chips. ROM chips. Overlay. Addressing problems. Start-up conditions. The ULA chips. The CRT controller. The memory map. Disc workspace.

3 **Binary and hexadecimal** 21

Digital versus analogue computers. Representing numbers by switches. Bits and bytes. Ways of arranging bits. Nybbles. Naming individual bits. Binary addition. Double byte numbers. Signed numbers. Two's complement numbers. Circle of signed numbers. Unsigned binary. Decimal and binary conversion. Hexadecimal. Converting hex to decimal. Adding hex numbers. Binary coded decimal. Adding in BCD format. Use of BCD. Logical operations. AND, OR and XOR.

4 **Entering and running programs** 38

Using an assembler. The HiSoft DEVPAC assembler. Listing source code. Assembler options. Pass 1 and 2 errors. Assembler directives. Object code. Using the BASIC loader. Saving and retrieving code. Saving source code. Loading source code. Saving object code. Renumbering source code. Executing object code. Assembler manuals.

5 **The Z80 registers** 53

Registers. Source and destination rules. The single length registers. Register pairs. The index registers. The stack pointer. Use of the stack. The program counter. The flag register.

6 **Commonly used instructions** 63

Instruction mnemonics and operands. Addressing modes. Implied addressing. Direct addressing. Immediate addressing. Indexed addressing. Register indirect addressing. Indexed addressing. Operation codes. Clock cycles. Bytes taken. Flags update. Load instructions. Arithmetic instructions. Logical instructions. Incrementing and decrementing. Stack operations. Conditional jumps. Unconditional jumps. Subroutine calls. Comparisons. Bit tests. Shift and rotate instructions. Operation code details. Relative jump bytes.

7 **Using resident firmware** 86

Free software. Jump blocks. Commonly used resident subroutines.

8 **Addition and subtraction** 92

Numerical range limitations. 8 bit addition and subtraction. 16 bit addition and subtraction. 32 bit addition and subtraction.

9 **Decision making and loop structures** 101

Testing for zero, non zero, less than zero and greater than zero. Simple loop structures. Double byte up-counting and down-counting. Preserving register contents. Branching according to sign. Using the BIT test.

10 **Multiplication and division** 118

8 bit unsigned multiplication. 16 bit unsigned multiplication. 16 bit signed multiplication. 8 bit unsigned division. 16 bit unsigned division. 16 bit signed division.

11 **Input and output** 139

String input. String output. Text output. Decimal input. Hexadecimal input. Decimal output. Hexadecimal output.

12 Parameter passing and introduction to resident system extensions 161

Parameter blocks. Fixed locations. Using the stack. Executing machine code from BASIC. Sorting of BASIC string arrays. Logging on an RSX. External command tables. Name tables.

13 Self relocation of subroutines and resident system extensions 185

Dynamic allocation of memory. Self relocation of object code. Location independent object code. Converting existing code to a self relocatable RSX. Programming the user restart (RST 6). Converting absolute addresses to relative. Look-up tables. String arrays Quicksort RSX. Loading and testing the RSX. Parameter passing from BASIC to an RSX. Executing an RSX from BASIC. RSX for sorting rectangular arrays. Using rectangular arrays efficiently. How BASIC organises a rectangular array. The RSX loader test program. Producing a binary file of an RSX.

14 Graphics and direct screen addressing 213

Using resident firmware routines. Typical action games. Organisation of screen memory. Mode zero. Shape tables for Mode zero. Placing a shape at screen coordinates X,Y. Addressing screen memory directly. Frame flyback. Moving a multicoloured shape. Moving multicoloured shapes independently. Co-ordinate blocks. Shape tables.

Appendices

Index 244

Why machine code? 1

It takes an extra effort to learn machine code, and to persuade you that it's worthwhile, this chapter discusses the properties of high-level languages such as BASIC, the various methods of entering programs and explains the differences between compilers, interpreters and assemblers.

Good points of BASIC

BASIC is the most popular computing language in present day use and, in spite of periodic abuse from the establishment, will probably remain so for some time. The reasons are not hard to find. With few exceptions, home microcomputers are designed with only one built-in computer language, BASIC, so it is understandable that the majority of home enthusiasts stick with it because they feel it to be the 'natural' language of the computer. There are other reasons, not least of which is the undeniable fact that newcomers to the computer revolution find BASIC not too difficult to pick up. The original designers of BASIC, John Kemeny and Thomas Kurtz of Dartmouth College USA, way back in 1964, purposely designed the language to suit those who wanted to use a computer in their work but were not prepared to study the rather difficult languages currently used by professionals at that time. Another advantage of BASIC is the relative ease with which program lines can be deleted or modified. In all fairness, we should point out that this advantage stems from the method of translation, rather than from the language itself.

Defects of BASIC

The above treatment has dwelt on the plus points of BASIC but there are some rather disagreeable features which continue to provide ammunition for the critics. One criticism is concerned with the speed of *execution*. Depending on type, complexity and pro-

gramming skill, some BASIC programs can take quite a while to display the 'answers' on the screen. A few seconds' (or minutes') wait is easily tolerated during the first weeks of ownership because the natural fascination with a new toy clouds the critical faculties. However, after the settling in period has ended, even a few seconds can become irritating so the relatively slow execution speed of BASIC must be accepted as a justifiable criticism. (Why BASIC is slow will be explained later in this chapter.)

Another criticism concerns the amount of memory taken up by a program written in BASIC. As each BASIC line is entered, the bytes squandered tend to mount up at an uncomfortable rate. Home micros, even Amstrad machines, are far from lavish in the amount of RAM left over for user's programs. A sizeable chunk is always purloined by the resident software for purposes which come under the euphemistic heading of 'work space'. The situation is made even worse by the fact that the full BASIC program (the *source code*) must reside in RAM, in addition to the variable list, while the program is being run. The strict definition of source code will be tackled later but, in the meantime, it can be thought of as the program in the form that it was originally written and entered. This is the form which squanders RAM. Most other languages only require the machine code translation (called the *object code*) to remain in memory during run time.

Although execution speed and memory demands are often quoted by attackers of BASIC, by far the most savage criticism is its lack of *structure*. To exponents of the art, structure, a normally harmless little word, is a kind of religion. It would be out of place in this book to enter into lengthy discussions on what constitutes good structure in a program because the concept is not quite so important in machine code work. It is sufficient here to describe it as a set of rules and guidelines for writing programs which are easy to follow and easy to amend. It is not easy to write good structured programs in BASIC because the language was designed without regard for structure. Many other computer languages such as Algol and Pascal were designed specifically for the writing of structured programs. Fortunately, BASIC has gradually changed since its conception in 1964. Dialects of BASIC, now appearing in some modern microcomputers are far removed from the original crude Dartmouth version. Both the BBC and the Amstrad machines offer quite an advanced BASIC, allowing better structured programs to be written. At least it is possible in either of these machines to write a program without too many GOTO commands polluting the pages. (The GOTO command is responsible for the cursed 'spaghetti' programming.)

In spite of all these criticisms, the authors do not despise BASIC. There is no reason why you should give up BASIC to learn machine code. In fact, the two can live quite happily together. One can complement the other in many ways. The abuse, now hurled at BASIC, cannot be accepted without suspicion as to the motives of

those who practise it – in the words of the Bard, they 'protesteth too much'.

Translating high-level languages

Languages such as BASIC, Algol, Pascal and many others (there are over 1000 in all if various specialised languages are included in the total) are said to be *high-level*. A high-level language:

(a) requires the assistance of *translation software* before the computer can understand it and,
(b) is one in which each statement, command or keyword in the language, when translated, usually gives rise to many individual machine instructions.

In BASIC, for example, an apparently simple statement like PRINT TOTAL will, after translation, generate a surprising number of machine code instructions. This is due to:

(a) the primitive nature of a computer at machine level and
(b) the necessity for the translating software to cover worst-case situations. For example, the variable TOTAL could be holding a small number like 47 in one program but a huge number like 358,467,789 in another so the translation software must always be prepared for the worst.

Compilers and interpreters

There are two quite distinct classes of high-level language translators and we need to distinguish them with care. Before describing the difference, we must first introduce the terms *source* code and *object* code.

Source code is the program as written in the original language. A program written in BASIC or Pascal is source code. Code in this form may be understood by you but not by the machine. Object code is the output from the translator. In other words, it is the set of machine code instructions which appear after the source code is translated. This time, the machine will understand it, but you may not.

Compilers

A compiler translates the entire source code into a *complete set* of machine code instructions. This is said to be the compiling stage and the source code is then said to be compiled. Once compiled into object code, the program can be executed (run). The original source code can then be released from occupying valuable RAM space. Pascal, Fortran, Cobol, Algol are examples of languages which are translated by a compiler. It takes a relatively long time for a machine to compile source code but it is only a one-off task. All subsequent

runs take place at machine code speed. From the beginner's point of view, a compiled language has one serious drawback. Once the program has been compiled, it is extremely difficult to debug or modify the object code. If it doesn't work first time, the source code must be re-entered before the fault can be corrected and the whole lot recompiled again which, as we have seen, is a lengthy business. Figure 1.1 shows the mechanism of compiler action.

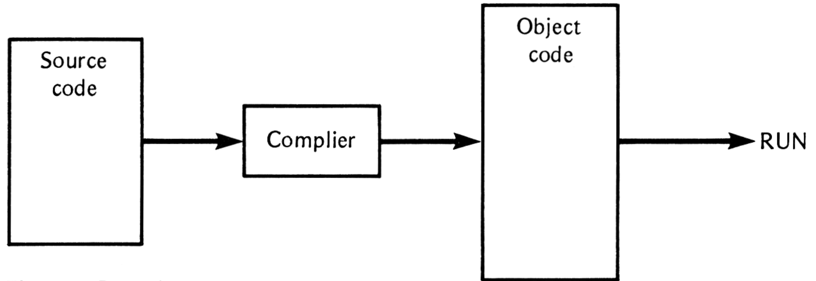


Fig 1.1 Compiler action

Interpreters

An interpreter does not translate all of the program in one go. Each line of the source code is translated and executed (run)) *before* the next line is translated. So the overall action is: translate a line, run it, translate the next line, run it, and so on. This means that the entire source code must remain in memory during run time. BASIC uses an interpreter. From the beginner's point of view, the advantage of an interpreter lies in the ease with which a program can be debugged or twisted around and re-run immediately. The disadvantages however, are serious. Since the source code must remain in memory during run time, valuable RAM space is occupied which limits the size of a program. Also the execution is dreadfully slow compared with compiler action because translation takes place every time the program is run. Figure 1.2 shows the mechanism of an interpreter.

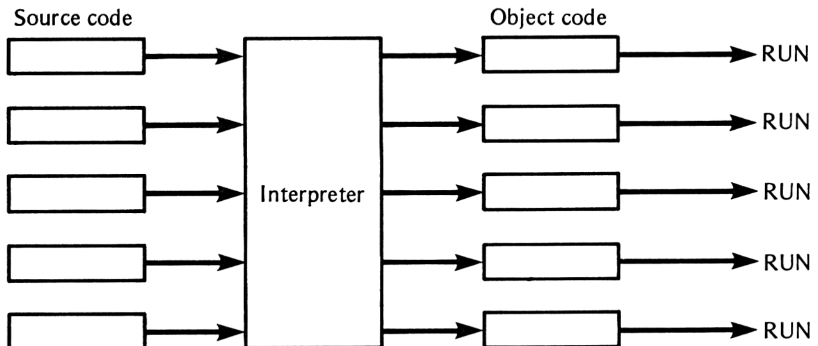


Fig 1.2 Interpreter action

Assemblers

As far as this book is concerned, compilers and interpreters are not particularly important. They were introduced only as background material before introducing the assembler. Compilers and interpreters are concerned with the translation of high level languages. An assembler is a piece of software designed to aid the writing of direct machine code. Detailed treatment of an assembler is given in Chapter 4. It is sufficient at this stage to briefly describe assembler action. High-level interpreters generate many machine code instructions for each high-level statement. In contrast, an assembly language program is said to be *low-level,* meaning that, in general, one assembly code line generates only one machine code instruction. We could say there is a one-to-one relationship between an assembly code line and a machine code instruction. The primary objective of an assembler is to ease the task of constructing a machine code program. Instead of entering a long list of almost meaningless numbers, an assembler allows you to enter them in the form of code letters chosen, as far as practical, to be meaningful. They are called *mnemonic codes* because they aid your memory. A good assembler, in addition to allowing mnemonic code entries, will also provide sophisticated editing facilities, display of both source code and object code, means of storing either the source or object code on disc (or cassette tape) and various other little dodges which help to make life easier.

The terms source code and object code still apply to assemblers. The source code is the program written in assembly language and the object code is the translated version recognisable by the machine. The object code is said to be the assembled version. To 'assemble' means to translate to object code.

Is an assembler necessary?

An assembler is not provided free with the Amstrad machines. It is possible to enter a machine code program by a series of POKES from BASIC. Later, we shall be giving a complete listing of a BASIC program which will load any machine code program you like to write, providing the *hex codes* are entered in DATA statements. So, it would appear from this that the answer to the question 'Is an assembler necessary?', is no. This needs amplification. If you have patience above normal, if you are not easily frustrated, if you are willing to devote hours and hours to debugging and, above all, if you are blessed with a stable unemotional personality, then by all means save yourself some money and make do with POKEing. If, on the other hand you are a normal individual and intend to take up machine code seriously then you are strongly advised to go out and get hold of an assembler. Trying to write machine code without the assistance of an assembler is like trying to cut a lawn with nail scissors.

The current success of the Amstrad has stimulated software firms

to produce assemblers on tape, disc or in ROM. There is probably not much to choose between any of them, price for price, but the programs in this book have been developed using the official version, the HiSoft DEVPAC assembler. Its full title is as follows:

The HiSoft Assembler, disassembler, editor and monitor

It can be obtained from AMSOFT or direct from HiSOFT. The price, at the time of writing is around £28. There are two versions, one for the CPC464 which is supplied in the form of a cassette tape and one supplied on disc for either the CPC464 with disc drive or the CPC664 and CPC6128 with built-in disc drives.

The advantages of using machine code

Before outlining the advantages, we should mention that 'machine code', is a rather loose term used to cover several different forms. There are in fact three different forms:

1. Pure binary machine code

A program residing in RAM would appear as a series of pigeon holes, each containing a set of '1's and '0's. A program in this form is called pure binary machine code. The very early computers had their programs entered in this way by means of a series of panel switches. Mercifully, those days are ended but only because more humane ways have been thought up for entering code. Nevertheless, computers still only understand these '1's and '0's – the change has been in the method of entering, not the form in which they are finally held in memory. Here is an example of a short program segment to clear the contents of memory address 0025 hex, as it might appear written in binary machine code:

```
1001 0111
0011 0010
0020 0101
0000 0000
```

(It is pointless at this stage trying to work out how this, and the next two examples work.)

2. Hexadecimal machine code

This is code entered in the form of a series of numbers which can be entered from a normal keyboard. A mixture of hardware and system software provides the necessary conversion to the equivalent binary '1's and '0's. Without an assembler, this is the form in which programs have to be entered. This is what happens when you POKE machine code or enter it by means of a loader. The numbers can be

in decimal or hexadecimal. Although decimal appears more natural for humans, hexadecimal will be found more natural for the machine and, with practice, easier for the human to communicate with it. Chapter 4 deals in detail with the hexadecimal system of numbers.

The program segment written in binary machine code shown above now follows as it would appear if written in hexadecimal machine code:

```
97
32
25
00
```

3. Assembly code

Assembly code is best considered as the most elegant and the least error prone method of entering machine code. When we use assembly code we still speak of it as machine code so from now on, the term 'machine code' and assembly code mean the same thing. As a final example, here is how the same program would appear if written in standard Z80 assembler code:

```
SUB A
LD (#25), A
```

Why learn machine code?

Quite understandably, anyone used to the comparative comfort of BASIC will not find machine code easy. Machine code is difficult, and it is best to understand what you are up against right from the start. There are many rewards for the extra effort involved. Let us list them:

(a) A machine code program *can run much faster* than one written in BASIC. Depending on the type of program, anything from 10 to 1000 times as fast.
(b) A machine code program takes up considerably less RAM space than the equivalent BASIC version.
(c) Learning machine code will improve your understanding of computers because you are conversing directly rather than through a wall of translation software.
(d) Animation routines, written in BASIC often appear jerky because of the slow execution time. Machine code versions give a greatly improved display. To drive this point home, we should take note of the fact that commercial games programs are almost always written in machine code.
(e) It is possible to recoup the money spent on a computer, plus some extra, by selling your programs to a publisher or software house. However, very few commercial organisations accept pro-

grams written in BASIC, a condition which applies particularly to animated games.
(f) Mastering machine code can act as an ego boost. A shallow advantage perhaps but few of us are entirely without vanity.

Mixing BASIC with machine code

Until you gain experience, you would be well advised to mix BASIC with machine code subroutines. Rather than attempt to write complete machine code programs, be content with writing BASIC programs which call up machine code at critical points. Use BASIC lines for actions which are not sensitive to execution time and machine code where its properties justify the extra effort. For example, in a general purpose database, much of the program will work quite satisfactorily in BASIC but where it is required to search for a particular record, or to sort them in to some form of order, use machine code subroutines.

The microprocessor instruction set

A program written in BASIC for one make of machine will not necessarily run on another make. Although the majority of the statements and commands remain the same for different machines, there will be a few variants. In spite of this, it is usually a fairly simple job to modify a BASIC program, written for one make of machine so that it will run on another.

With machine code the program must be written to suit the particular type of *microprocessor* used, not the machine system as a whole. The microprocessor is the chip which forms the so-called 'brain' of a microcomputer and it will only understand machine code written in its own special language. For example, a program written in BASIC for, say the Commodore 64, can be persuaded to run on the Amstrad or the BBC machine, providing a few trivial changes are made. However, a program written in machine code for the Amstrad would have to be completely re-written before it could work on the Commodore 64 or the BBC machine because the Amstrad uses the Z80 microprocessor and the BBC and the Commodore 64 machines both use the 6502/6510. Studying machine code for the Amstrad, or any other machine which uses the same microprocessor, will involve learning the *instruction set* of the Z80. The instruction set of a microprocessor is the list of machine code instructions which it is capable of executing, together with the hexadecimal or mnemonic codes which it recognises. It is a formidable looking list of over 600 different instructions. Nobody in their right senses would attempt to memorise them all, in any case only a small percentage of them are in regular use. As long as you can learn how to read the instruction set, that is all that matters.

Summary

1 BASIC is relatively easy to understand but uses a lot of memory and is slow in execution.
2 BASIC, PASCAL and ALGOL are examples of high-level computer languages.
3 Each statement in a high level language is equivalent to a great many machine code instructions.
4 High-level language translators are either compilers or interpreters.
5 A compiler must translate the entire high-level program into machine code before it can be executed.
6 An interpreter translates, then executes, line by line.
7 Source code is a program in the original form. Object code is how it appears after translation to machine code.
8 An assembler is software designed to aid the writing of machine code.
9 Without an assembler, machine code can only be entered by POKE statements from BASIC.
10 Machine code executes rapidly and is economic on memory.
11 Machine code programs must be written to suit the microprocessor in use. In the case of the Amstrad, this is the Z80.

2 Machine hardware

The hardware and memory usage of the CPC 464 are described here, but much of the material is still valid for the later CPC664/6128 models.

Is hardware important?

In some respects, a computer is no different from a washing machine. Providing you know how to use it, knowledge of what's inside is of little importance. Certainly, if your use of a computer is restricted to buying and running commercial software there is no need whatsoever to concern yourself with hardware details. To a slightly lesser extent, you can maintain the same lofty indifference if you write your own programs in a high level language such as BASIC. After all, the general idea behind any high level language is to protect the user from the harsh realities of computer technology. A BASIC interpreter does just that by forming a cosy software blanket between the user and the machine. But the machine code enthusiast is normally a different kind of animal, who wants an in depth understanding, even if it does entail additional mental labour. The question now arises as to the level at which hardware needs to be studied by the newcomer to machine code. For example, considering that a computer is virtually 100% electronic in nature, is it necessary to study electronics? Fortunately for most of us, the answer is no. It may be useful to have a superficial electronic awareness but, unless you intend to go in for computer repairing or designing as a side line, you can get away with little more than lamp, battery and switch knowledge. Gone are the days when the circuit diagram of a computer stretched over hundreds of pages, each filled with resistor, capacitor and transistor symbols. All this complexity ended with the introduction of the *integrated circuit*, back in the middle sixties. In a modern home computer, such as the Amstrad, over 90% of the complexity is buried inside a handful of integrated circuits, ICs. The machine code enthusiast need not bother with the insides of the ICs. All he needs is a rough idea of the

function of each major IC and how they are connected together to form a computing system. The highly simplified version of the system can then be reduced to one or two *block diagrams*. That is to say, diagrams containing 'boxes', representing the ICs, connected to bundles of wires called *buses*. A bus is the term used for a set of wires, all conveying the same kind of information. The *address* bus, the *data* bus and the *control* bus are concepts which will be mentioned frequently during this chapter. However, before continuing, we should make it plain that information given in this chapter is not absolutely essential for the machine code programmer. You don't have to understand what follows. All we can say is that the knowledge will not be wasted. It will help you appreciate the finer points of the art and, more importantly, it will help you understand what is actually happening in the computer when you later write, say,

LD A, (HL).

Logic chips

Logic is a much abused term. In everyday speech, it usually means clear thinking. In computer language, it means circuits which can only be in *one of two states*, called either ON or OFF, or 1 and 0. Apart from a few exceptions, all wires on the computer board are either in the '1' state (voltage around three to five volts) or the '0' state (voltage around zero to one volt state). Complex logic systems are based on a few building bricks known as *logic gates*. These have a single output and, normally, several inputs. The output of a logic gate depends on the right combination of '1's and '0's being present on the inputs. ICs may contain hundreds or even thousands of internal logic gates, all interconnected to form an overall subsystem. All subsystems are synchronised by an electrical oscillator known as the *clock*. The clock is driven by a *quartz crystal* in order to maintain frequency stability. In the Amstrad, the clock frequency is maintained at 16 million oscillations per second (16 MHz).

The Z80A microprocessor

There is one particular chip in the Amstrad, and indeed in all home and personal computers, which actually does the computing. This chip is called the *microprocessor* and, in the Amstrad CPC464/664/6128 machines, it bears the type number Z80A. All other chips in the system are subservient to it. It is capable of responding to outside instructions which 'tell' it whether to add, subtract, fetch some data from memory etc. The instructions must be given in patterns of '1's and '0's because it only understands *binary* and can only perform with binary numbers (Chapter 3 treats binary in detail). Needless to

say, it doesn't understand a word of English. Furthermore, it can only respond to instructions which are within a limited repertoire known as the *instruction set*. The Z80A has a particularly rich set of over 600 instructions. You may think this is a bewildering number from which to pick out the one you want but when grouped together, according to respective functions, we find that those in the same group are merely subtle variations on a main theme. All data into or out of the Z80A takes place via *registers*. The Z80A is well equipped with registers, details of which are in Chapter 5.

Figure 2.1 is our first block schematic. It is a highly simplified version of the Amstrad circuit board intended only to illustrate the central position of the microprocessor in relation to the general scheme of things.

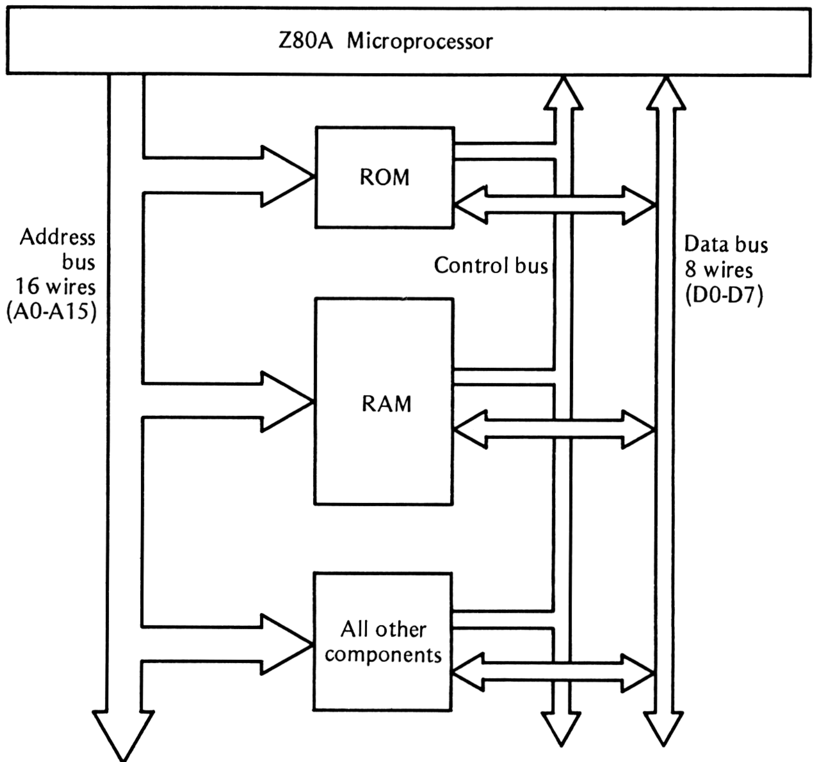


Fig 2.1 The microprocessor and buses

All information between the microprocessor and the rest of the main components (lumped together as one box) take place via three buses.

The address bus

Each memory cell (and certain auxiliary components) is identified by

a unique binary code known as its *address*. So when a particular cell is to be activated, the microprocessor sends out the personal code of this cell down the address bus. Cells ignore all address signals other than their own. There are two important details to note:

1 The direction arrows on the diagram show that the address bus is *one-way*. Address codes can only originate from the microprocessor and travel *down* the bus.
2 There are 16 wires in the bus, (labelled A0 to A15) so there is an upper limit on the number of different addresses. Chapter 3 gives reasons why this limit can not exceed 65,536 different addresses. Decimal numbers such as this are not easy to memorise. Large numbers are best expressed in terms of 'K'. One K in computer jargon is 1024 (2^{10}) not 1000. In termsof K, the number 65,536 is reduced to a nice round figure of 64K. ($64 \times 1024 = 65,536$.) The lowest address is address 0 and the highest is address 65,535. We shall learn in Chapter 3 that, when working in machine code, addresses are far better expressed in the *hexadecimal* (hex) number system. In terms of hex, the address range is from 0000 to FFFF.

The data bus
This is an 8-wire bus (labelled D0 to D7) which carries *data* between the microprocessor and the rest of the system. All 8 bits (known as *one byte*) are transferred simultaneously along the data bus. Note that the direction arrow, unlike the one for the address bus, is *two-way* which means that data can pass to the external devices (called *writing*) or from the external devices (called *reading*). The fact that the data bus is 8 bits wide, brands the Z80A as an '8-bit' microprocessor because only numbers which can be held within the compass of 8 bits can be transferred in one go. The highest absolute decimal number using 8 bits is only 255 so larger numbers must be fetched in 8-bit instalments.

At this point, perhaps we should mention 16-bit microprocessors which are heralded as forming the new generation of home computers. A true 16-bit microprocessor, apart from whatever other qualities it may have, is one which has a full 16 bit data bus. A few so-called '16-bit' machines which are now on the market do indeed have microprocessors which, internally at least, operate on 16 or even 32 bit numbers simultaneously. But they employ one of the cheaper versions of the 16 bit family which only have an 8 bit data bus. This means that 16 bit data still has to be transferred in two instalments.

The control bus
The control bus, unlike the address and data buses, is a bundle of odds and ends, a hotch potch in fact. Each wire has an entirely distinct function. Some carry orders from, and some carry requests to, the microprocessor. For example, one of them will be the read/write control line, another will carry the clock signal and another will be the reset line.

Figure 2.2 gives a more detailed picture of the microprocessor and components, and is an advance on the previous block schematic

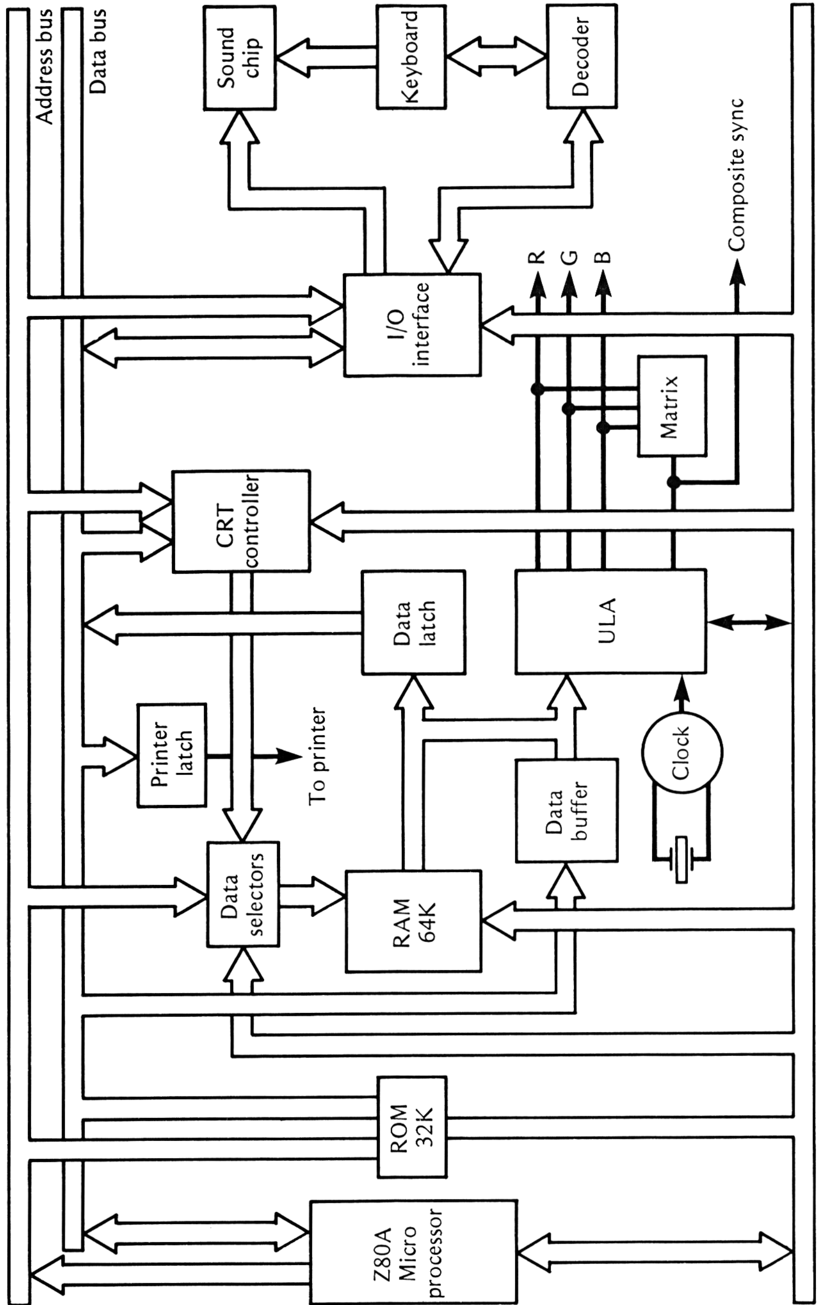


Fig 2.2 The block schematic of the Amstrad 464 and 664

although even this is still an enormously simplified version of the true picture. Nevertheless, the detail shown is more than adequate for our purpose.

The clock
This runs at 16 MHz for direct drive to the ULA (see later) but counted down by a factor of four to drive the microprocessor at 4 MHz. However, a slight complication arises because of the need to synchronise with the video scanning circuits. This has the effect of reducing the effective clock rate to 3.3 MHz.

The RAM chips
The total complement of read/write memory is contained in *eight identical chips*, each capable of storing 64K separate data bits. Because there are eight of them and because each contributes one bit of data, they collectively act as a 64K byte memory. The address inputs of the RAM chips are fed from the address bus, and the eight separate data wires are connected to the D0 to D7 lines of the data bus. Figure 2.3 illustrates the RAM chip connections.

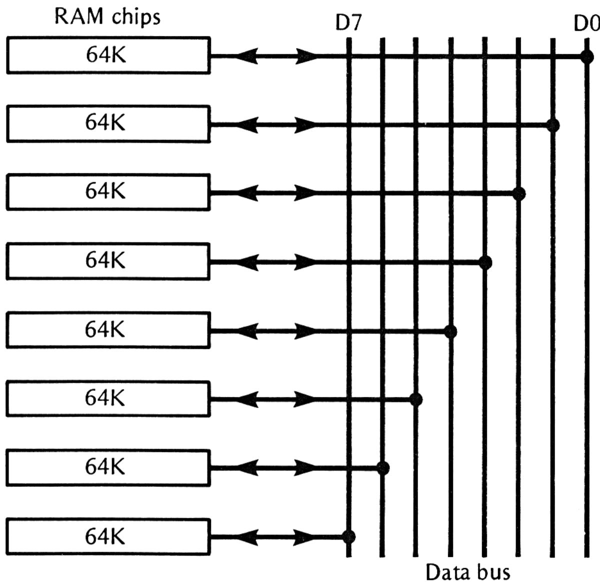


Fig 2.3 RAM chip connections

One of the lines (labelled R/W) in the control bus is also connected, albeit indirectly, to the RAMs for controlling the *direction* of data flow. That is to say, whether data is to be read from or witten to RAM.

The ROM chip
The single ROM chip holds 32K of permanent programs. Programs

held in ROM are known under the collective title of *firmware* rather than software. Although in a single chip, the ROM should be considered as two functionally separate 16K ROMS, the 'lower' and the 'upper'.

The *lower* ROM occupies the address range 0000 to 3FFF hex and contains the *Operating System.* Duties under the heading of the operating system include reading the key board, handling peripherals, sending information to the screen display circuitry etc. In short, all the various mundane duties which help to provide a friendly man/machine interface. The success of any machine, whatever other qualities it may have, can stand or fall on the quality of the operating system. Only programmers of high calibre are employed in writing operating system firmware. Needless to say, firmware is written in machine code.

The upper ROM contains the BASIC interpreter firmware and occupies addresses C000 to FFFF hex. It contains all the translation rules for converting BASIC programs into machine code. The quality of the Amstrad BASIC interpreter is excellent.

Addressing problems

Most home computers take the easy way out and restrict the total RAM and ROM complement so as to fit comfortably into the available 64K addressing space of 8-bit microprocessors. (You will remember that a 16-bit address bus can only supply 64K different addresses.) But, Amstrad has 64K of RAM and 32K of ROM! This apparent violation of the laws of address combinations obviously calls for some explanation. The problem is overcome by bank switching. The upper and lower ROMs 'overlay' the RAM areas beneath them. That is to say, the areas of RAM, which underlay the ROMs, share each other's addresses. This is all very nice but an obvious problem remains. Since parts of RAM and ROM share the same addresses, how does the system distinguish between them? The problem is sorted out by the mysterious ULA but is helped by the following reasoning:

1 If the instruction is to write to memory, it must mean the RAM because you can't write into a ROM.
2 If the instruction is to write or read to any address between 4000 and BFFF hex, it must be RAM because there is no ROM in this area.
3 If it's reading information from ROM areas, it can be from either ROM or the underlying RAM. This is where the ULA takes over and decides whether the ROM should be isolated, leaving the coast clear for the RAM or vice versa.

The above description has been based on the normal (default) conditions of the Amstrad. However, it is possible to modify this arrangement by a certain machine code call. Thus it is possible to

disable one or both ROMs although such adventures are best left until your experience has matured a little.

Start up conditions
On first switching ON, the lower ROM takes over, resets all devices and clears all RAM locations. On completion, control switches to the upper ROM, ready for the first BASIC command.

The ULA
Computers, up to a few years ago, used to bristle with logic chips, most of which performed primitive gating functions. Reliability is inversely proportional to the chip count so designers were constantly searching for ways to reduce the number without sacrificing sophistication. One ingenious solution was the *Uncommitted Logic Array* (ULA for short). The idea is to construct a large number of separate gates on one chip with provisions for 'burning in' the interconnections between them afterwards. This means that the chip was initially undedicated to any particular function. The actual burning in stage is left until the end user supplies a plan of the final interconnections. When this is done, the once uncommitted ULA becomes committed to undertake a complex but specific task. It becomes a customised chip, capable of replacing a large number of untidy separate chips. The ULA solution is commercially viable for bulk orders in excess of, say, 20,000. The cost per chip of the final burning in process is excessively high for small quantities.

The ULA in the Amstrad is labelled IC 116 on the circuit diagram and is responsible, amongst other things, for the following:

(a) Dividing the clock frequency by four to feed the Z80A.
(b) Controlling the data selectors which switch the address bus between the video controller and the Z80 on a time ratio of 3 to 1.
(c) Indirectly controlling the colour of the screen output characters by means of a *palette latch*. The input to the palette latch is derived from the data bus (because the programmer decides the colour) and the output is in the form of a five bit code. Decoder circuits in the ULA reduce this to the three Red, Green and Blue (RGB) wires which feed the colour monitor. There is one mystery here. It is possible to program up to 27 different colours and yet there are only three wires! The answer is to feed them with *tristate* instead of *two state* logic. Thus, each of the separate RGB outputs have been arranged so that they can rest in any one of three states, logic 1, logic 0 or virtually open circuit. Three wires, each capable of resting in any one of three states, are capable of $3^3=27$ combinations.
(d) Periodically switching in the Cathode Ray Tube Controller (CRTC) to the screen RAM. It does this by *interrupting* the microprocessor every few instruction cycles.

The CRT controller
This is a standard commercial chip bearing the type number 6845. It

is quite a complex affair with eighteen internal registers and is almost as flexible as a microprocessor. It is a *peripheral*, in the sense that it has a port address block which can be activated by the BASIC instructions INP and OUT. However, most of the CRT controller registers are loaded by the operating system ROM and require no help from us. It performs, by hardware, a large number of screen operations, including scrolling, sync, high-res graphics and text positioning. A less complex CRT controller would require extra software in the ROM to achieve the same control of the screen.

Data selectors

Referring to Figure 2.2, the block marked 'data selectors' consists of 4 chips, responsible for steering the data bytes to RAM from either the Z80A or the CRT controller. The chips act like 8-way, two-pole switches controlled by the ULA.

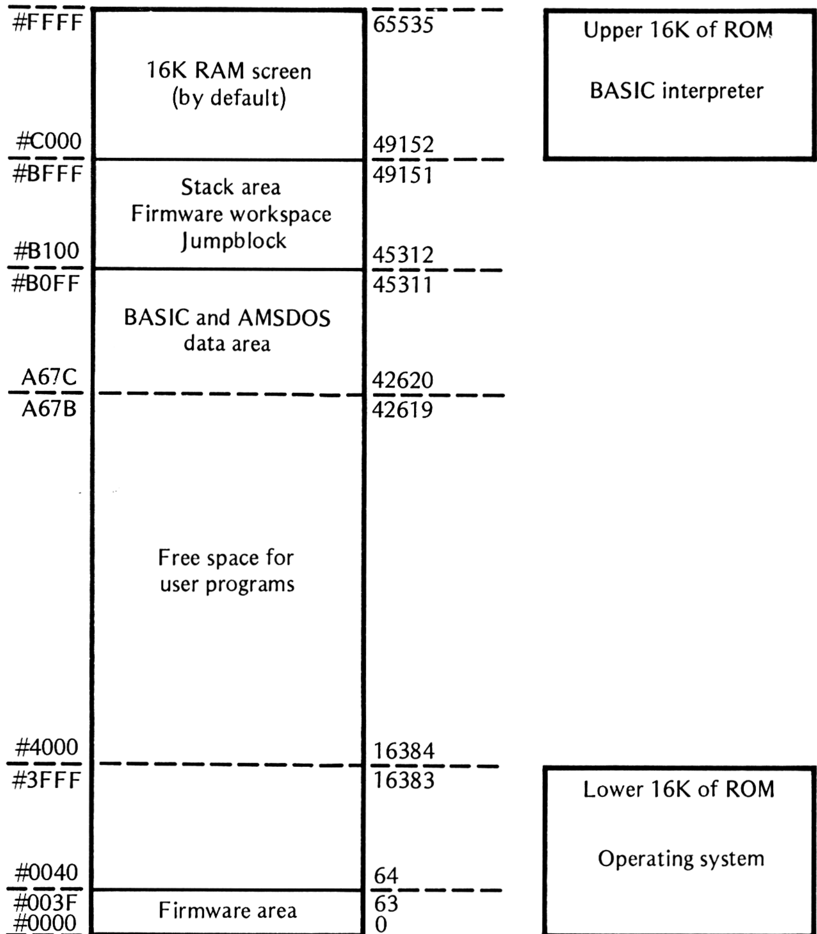


Fig 2.4 Memory map

The memory map

The memory map of the Amstrad is shown in Figure 2.4. For convenience, the memory addresses have been given in both decimal and hex. To indicate hex numbers, we have used the character '#' because it conforms to the notation used by the Hisoft DEVPAC assembler.

The map shows the areas in which the upper and lower ROM addresses overlay RAM. Note the *stack* (explained in later chapters) is left by the operating system in a position below the screen RAM area.

The sound generator

The Amstrad sound effects are produced by the industry standard AY-3-8912 chip. It is connected via the input-output interface. It is possible to program the hardware registers direct but it would be a dangerous procedure because of its close linkage with the keyboard scanning. This is a highly critical area because of precise interrupt timing.

Disc workspace

The Amstrad CPC664/6128, or the 464 fitted with a disc drive, requires an extra 1284 bytes of RAM for the disc operating system (DOS) workspace. This means that HIMEM will be lowered from #ABFF (44031) down to #A67B (42619). Some lengthy programs written for the 464 on tape may not work if a disc drive has been added. The solution is to leave the disc drive switched off. On the 664, the DOS is always on so the simple solution won't work. However, an excellent software patch for overcoming the problem has been designed by Cliff Lawson (see 'Amstrad User' magazine, July 85 issue).

Summary

1 A bus is a bundle of wires, each conveying the same kind of information.
2 The 16 wire address bus carries the code for selecting one particular memory cell.
3 The 8 wire data bus conveys data bytes around the system.
4 The control bus contains a mixture of wires (so it is not a true bus) each having a specific control function.
5 Logic, in computer parlance, refers to any two state switch system. The states can be called 1 and 0, ON and OFF or HIGH and LOW.

6 A logic gate has one output, the state of which depends on the logic combination applied to the inputs.
7 A 16 MHz clock is the source of all timing pulses in the Amstrad machines.
8 The clock frequency is divided by 4 to obtain the 4 HMz drive to the Z80A microprocessor.
9 Due to the requirements of video scanning, the effective Z80A clock frequency is reduced to 3.3 MHz.
10 The read/write memory consists of 64K of RAM.
11 The read-only memory is a 32K pre-programmed ROM.
12 Permanent programs in ROM are known as firmware.
13 The upper 16K of ROM contains the BASIC intepreter firmware and the lower 16K contains the Operating System.
14 A 16 wire address bus can only manage 64K different address combinations so ROM addresses overlay some RAM addresses.
15 The upper ROM overlaps the screen RAM (C000 to FFFF hex.)
16 The lower ROM overlaps the operating system workspace (0000 to 3FFF hex).
17 The ULA is a customised chip responsible for a range of assorted switch actions.
18 The RAM screen area, the upper ROM and the CRT controller all share RAM addresses. The ULA arranges the periodic switching at speeds transparent to the user.
19 The ULA also takes care of the colour palette decoding, mode information, and composite sync generation.

Binary and hexadecimal 3

Digital versus analogue computers?

The term 'computer' is well established. So we are inclined to forget that there are two species and that when we speak of the computer age we are automatically referring to the *digital,* rather than the *analoge,* species. And yet, the analogue computer came first mainly because the physical characteristics of the real world can be modelled more naturally by analogue techniques. Ignoring sudden volcanic eruptions or similar violent catastrophies, physical changes in the real world take place in a smooth orderly manner. For example, wind speed will not suddenly jump from 10 mph to 20 mph. A plot of windspeed against time will always be a *smooth curve.* It may be a steep curve during the onset of a storm but it will still be smooth providing, of course, that enough plotting points are taken. The same can be said of temperature changes. Even the dramatic heat rise at the beginning of a nuclear explosion may appear instantaneous but the temperature plotted at nanosecond intervals would still show a steep, but nevertheless smooth, curve. Nature, apart from some rather weird 'quantum' behaviour at subatomic levels, seems to dislike sudden jumps. Measurement of physical quantities is carried out with instruments which convert them to an analogous form, which we will assume here to be electrical, and are therefore known as *analogue* instruments. These instruments give readings, usually in the form of a voltage, which are, as far as possible, *proportional to the physical quantity.* For example, a wind velocity instrument may be calibrated such that each additional volt represented an increase of 5 miles per hour. But whatever the physical quantity, the essential ingredient of any electrical analogue device is that a certain voltage (or perhaps current) corresponds to a particular value of the physical quantity. In the case of our example wind measuring instrument, 1 volt= 5 mph, 2 volts=10 mph and 20 volts= 100 mph. How can we use analogue methods for manipulating numbers? In other words, how is an analogue computer constructed, assuming it is capable of addition, subtraction, multiplication, division and all the common-

place trigonometrical operations. All we need is a panel full of knobs which we can adjust to represent the numbers, a collection of fairly straightforward electrical circuits which can add, subtract etc and a few meters to present the results in a visual form. Indeed, this is exactly what a commercial analogue computer looks like. Such instruments still have a place in laboratories and are superior in some respects to the digital computer, particularly in their ability to handle complex differential equations. They have two drawbacks. Firstly, they require highly skilled operators to set up all the knobs and to check the preliminary reset conditions. Secondly, the analogue circuitry requires frequent adjustment to counteract voltage drifting caused by heat changes. The accuracy of an analogue computer relies absolutely on the accuracy of a voltage or current reading. Even with modern circuitry, equipment designed for voltage tolerances better than 1 part in 1,000 would be quite costly. This means that any attempt to use an analogue computer to work out a balanace sheet for a business organisation would fail the first auditor's check.

A digital computer overcomes all problems caused by measurement errors. In fact, the term 'accuracy' doesn't really apply to the digital computer because measurement is not part of its nature. Instead of *measuring*, the digital computer *counts*. To humans, counting can be tedious and error prone but a machine built to count, rather than measure, can be foolproof. The circuits of a digiatal computer need only be designed to detect whether a voltage is above a certain upper threshold limit or below a certain lower threshold limit. In other words, any individual circuit is always in the 'LOW' state or the 'HIGH' state. In the case of the Amstrad and most other current microcomputers, the two levels are approximately as follows:

> Voltage between 3 to 5 volts is the HIGH state.
> Voltage between zero to 1 volt is the LOW state.

The terms HIGH and LOW were chosen to suit electronic engineers but, from a computing point of view, it is better to adopt the terms 1 and 0 respectively. Figure 3.1 may help you to visualise these states.

Representing numbers by switches

Almost all circuits in a digital computer act as electronically operated *switches*, capable of changing rapidly from one state to the other on receipt of an appropriate signal. A series of switches can be made to represent numbers providing we are prepared to dispense altogether with the familiar decimal system and adopt the primitive numbering system known as *binary*. Decimal is out because it would

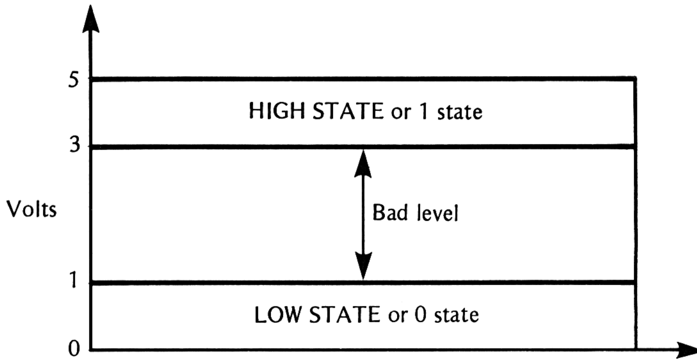


Fig 3.1 The two digital states

require a ten state electrical system. In other words, it would be necessary to revert back to an analogue system in which each circuit would have to respond to ten different levels of current or voltage. This demands expensive circuitry and frequent calibration. There are no calibration problems with a switch because it is either ON (the 1 state) or OFF (the 0 state)

Bits and bytes

Only the digits 1 and 0 are allowed in the binary counting system so it is inherently simple. The term bit, derived from binary digit, refers to either a 1 or 0. A string of eight bits is nowadays called a byte. The numerical value of each bit is based on 'place weighting', similar to decimal. However, each bit is worth twice as much (instead of ten times as much) as the bit on its right. For example, the byte 00001111 is equal to 15 in decimal because, proceeding from right to left, we have a 1 and a 2 and a 4 and an 8. Study the following examples:

00001001 = 9. 00000101 = 5. 00101110 = 46. 11111111 = 255. 10000000 = 128.

Number of ways of arranging bits
With two bits, there are 4 possible patterns, 00, 01, 10 and 11. With 3 bits, there are 8 patterns, 000, 001, 010, 011, 100, 101, 110 and 111. Instead of laboriously writing down all the possible arrangements of bits in a given string, we can use the following general rule:

Number of ways=2^N (where N=the number of bits)

For example, for 8 bits, there are 2^8=256 ways. Every time we add one more bit to the string length, we double the possible number of arrangements.

Nybbles

It is convenient to consider the byte in two halves, each known as a nybble and to introduce a space between them. For example, 0010 1101 is easier on the eye than 00101101 although it is important to remember that the space is quite artificial and has no arithmetic meaning.

Naming individual bits

When referring to individual bits in a byte, the terms illustrated in Figure 3.2 are well standardised. The bit on the extreme right, named bit 0, is the *least significant bit* (1sb) and the bit on the extreme left, named bit 7, is the *most significant bit* (msb). Note carefully, because there is a chance of confusion, that the bits are numbered 0 to 7, not 1 to 8.

Binary addition

The process of adding two binary numbers together is essentially the same as we use for adding decimal numbers. In decimal, when the sum of a column exceeds 9, we carry a 1 over to the next higher significant column. In binary, a carry is made whenever the sum of a column exceeds 1. Rather than explain the procedure in detail, it is eaier to study a few examples, using complete bytes.

To add 1 to 1:

```
        0000 0001=1
        0000 0001=1
        ---------
Sum     0000 0010=2
        ---------
```

To add 3 to 5:

```
        0000 0011=3
        0000 0101=5
        ---------
Sum     0000 1000=8
        ---------
```

To add 17 to 31:

```
        0001 0001=17
        0001 1111=31
        ---------
Sum     0011 0000=48
        ---------
```

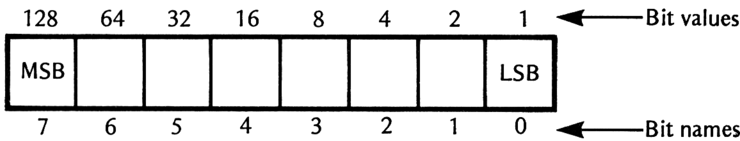


Fig 3.2 Naming the bits

Double byte numbers

The Z80 micro processor is an eight-bit device. This means that all trips to and from memory take place in bundles of eight bits. This is why, up to now, we have concentrated on the byte. The trouble with the single byte is the limited storage capacity. The maximum binary number (every bit a 1) which can be held in one byte is only equivalent to 255 decimal. Obviously, a computer would not be much use if it could only handle numbers up to this value. The way out is to use two or more bytes for each number by considering them to be arranged *end to end*. Assume we use two bytes and fill them both with '1's as follows:

1111 1111 1111 1111

This is equal to 65,535 decimal. You can of course check this by adding up the value of each binary digit but there is an elegant formula which, providing you have a scientific calculator handy (or the Amstrad in direct mode), can do the job quickly,

Largest number in a string of N bits is 2^N-1

For example,

4 bits can hold $2^4-1=15$
8 bits can hold $2^8-1=255$
16 bits can hold $2^{16}-1=65{,}535$
20 bits can hold hold $20^{20}-1=1{,}048{,}575$

When using double byte numbers, the least significant half is known as the *low-byte* and the most significant half, the *high-byte*. Note that each bit in the high byte has a value 256 times that of the equivalent bit in the low byte. This morsel of knowledge is of far reaching importance and justifies the appearance of Figure 3.3.

If the two bytes together are considered as one number the decimal value of the total is given by:

$$\text{Total}=(\text{low byte value})+(256\times\text{high byte value})$$

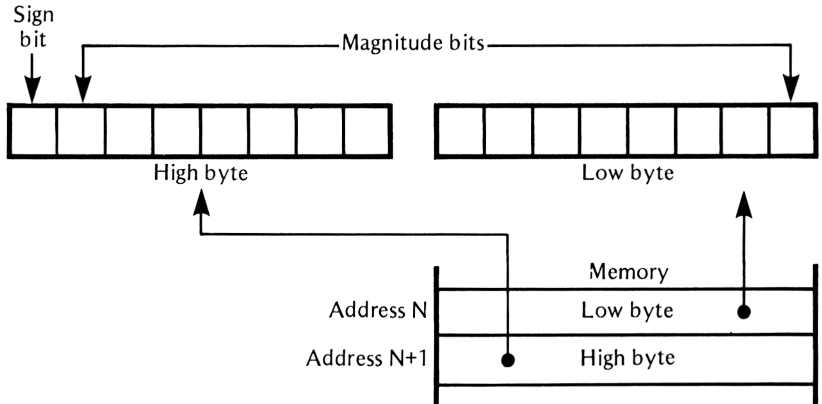


Fig 3.3 Equivalent bit values

If the low-byte holds 0000 0100 (4 decimal) and the high-byte holds 0010 0100 (36 decimal). The total becomes 4+(256×36)=9,220 decimal.

Signed numbers

A computer would be of little use if it could only handle positive numbers. Arithmetic used by humans uses the unary minus character '–' to indicate that the number is negative. This is not possible in binary because, as we have seen, the only characters recognised are 1 and 0. The Z80A microprocessor, in common with most other types, employs a system known as *two's complement* notation for dealing with signed numbers and, as an indirect consequence, subtraction. Before delving into the details of two's complement, it is worth devoting some space to the hardware mechanism of subtraction employed by most microprocessors. The only true arithmetic operation built into the Z80A microprocessor is addition. It can, of course, perform subtraction but, were we to examine the inernal hardware, we would discover that it still uses the additioin process but in a roundabout way by adding the two's complement of the number to be subtracted. This is because every square micrometre on the silicon chip area is precious. In fact the old Silicon Valley boffins used to refer to the area as 'real estate'. They decided that to include a hardware subtraction circuit in a microprocessor, as well as an addition circuit, would be a waste of chip area which could otherwise be used for more rewarding functions.

Two's complement numbers

The fundamental difference between normal binary (which we shall refer to as unsigned binary) and two's complement is the role of

bit 7, the msb. This bit no longer represents magnitude. Instead, it is used to indicate the sign of the number and, indeed, is often referred to as the *sign bit*. The sign bit is 1 for negative numbers and 0 for positive numbers. Although quite straightforward for positive numbers, there is a rather strange twist involved when the number is negative. Before giving the rules, consider the following examples of what positive and negative numbers would look like in two's complement notation:

0000 0010=+2
1111 1110=−2

As you can see, +2 is easy but −2 looks a horrible mess. You may ask why, since the msb is the sign bit, why not write −2 as 1000 0010? This would certainly appear more logical (and certainly easier) but there would be a snag when writing zero. For example, +zero would be written as 0000 0000 and −zero as 1000 0000. In other words, we would have two different ways of writing the same number. In any case, mathematicians have decreed that zero must always be positive so you see the simple way doesn't work.

The rule for two's complement numbers
First we must introduce the jargon term 'flip'. To *flip* a bit means to change it from 1 to 0 or from 0 to 1. Either of two rules can be used but the following is the accepted academic version:

> To obtain the equivalent negative, flip all the bits and add 1.

If, as the result of adding the extra 1, a carry propagates through to the end and 'drops out' then simply ignore it. Some examples based on the rule now follow.

1 To find −2, first write down the binary for +2 which is,

	0000 0010=+2
flip the bits	1111 1101
add 1	1111 1110=−2

2 To find −8,	0000 1000=+8
flip the bits	1111 0111
add 1	1111 1000=−8

3 To find −1,	0000 0001=−1
flip the bits	1111 1110
add 1	1111 1111=−1

We have said that this rule is the academic version. Flipping the bits produces the *one's complement* (also called the 'logical' complement)

and adding the extra 1 converts it to the two's complement. However, this rule may have the blessing of the establishment but adding that extra 1 is a nuisance because of keeping track of the carry. There is another way of finding the equivalent negative which does not involve messing around with that awful carry:

Starting from the right, copy down up to and including the first 1 then flip the remaining bits.

Examples:
+2=0000 0010
−2=1111 1110
+8=0000 1000
−8=1111 1000
+17=0001 0001
−17=1110 1111
+1=0000 0001
−1=1111 1111

Two's complement numbers work *both ways*. That is to say, the rules remain valid for finding the equivalent positive, given the negative version. For example, if we start with 1111 1111 (which is −1) and use the rule, we obtain 0000 0001 (which is +1 again). Thus the two's complement of a number doesn't necessarily mean the negative version. It simply means the same number with opposite sign.

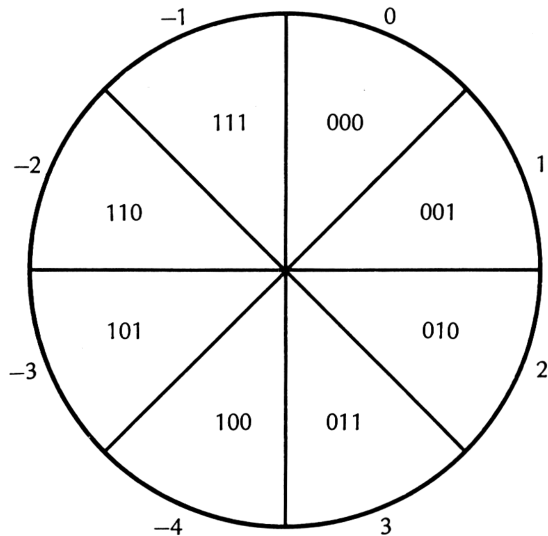


Fig 3.4 The two's complement circle

The circle of signed numbers

Although two's complement notation seems weird and even illogical, it is, in fact, based on sound number theory. To appreciate this, have a look at Figure 3.4 which, for simplicity, examines all the eight ways of arranging 3 bits. Note the following points:

(a) The binary combinations proceed smoothly from 000 to 111 as we journey clockwise round the circle. Thus, if we choose to treat the meaning in terms of unsigned binary, the combinations progress from 0 to 7.
(b) If we treat the meaning in terms of two's complement, we progress clockwise for positive numbers up to +3 or anticlockwise for negative numbers up to −4.
(c) There are 4 positive numbers, 0, 1, 2, 3 and four negative numbers −1, −2, −3, −4.

Because zero is a positive number, it follows that there will always be room for one larger negative number than positive in two's complement notation. For example, the largest positive number in a byte is 0111 1111=+127 but the largest negative number is 1000 0000=−128. To allay possible criticism, we should point out that mathematically, a number like −128 is considered to be 'smaller' than −127 but playing lip service to the demands of academic purity often hampers initial understanding.

Two's complement and double byte numbers

When dealing with double byte numbers, bit 7 in the low byte has no significance other than representing part of the overall magnitude. Only bit 7 of the high byte is the true sign bit. For example, consider the following double byte number, 0001 0000 1000 0000. The low byte has a 1 in bit 7 position but it does not represent sign because the number, taken in its entirety, is positive because bit 7 in the high byte is a 0. The largest double byte positive number, treated in two's complement form, is 0111 1111 1111 1111=+32,767 and the largest negative double byte number is 1000 0000 0000 0000= −32,768. On the other hand, if we treat the numbers as unsigned binary, the largest two byte number becomes 1111 1111 1111 1111=65,535. Perhaps this would be a good point at which to introduce a few equations for finding the maximum numbers that can be held in a string of N bits.

Largest positive two's complement number=$2^{(N-1)}-1$
Largest negative two's complement number=$-2^{(N-1)}$
Largest positive unsigned binary number=$2^{N}-1$

Unsigned binary or two's complement?

Readers may be a little confused at this point as to whether the microprocessor works in two's complement or unsigned binary. The

answer is simple. It depends entirely on your own intepretation of the result. If, for example, you are writing a program which is dealing with numbers that must always be positive (such as vibration frequency or population) then you would interpret the result as unsigned binary. On the other hand, programs concerned with banking accounts, must allow for both credit and debit situations and both positive and negative numbers. In which case numbers would be interpreted in two's complement form. As far as the microprocessor is concerned, arithmetic operations are carried out exactly the same, irrespective of the human interpretation placed on the results. To see why this is so, consider how the processor would add 1 to 0111 1111:

	0111 1111
Add	0000 0001
Total	1000 0000

Now, as far as the microprocessor is concerned, the job is complete. However, if the programmer is working in pure binary, the result is valid and equal to 128 decimal. If, on the other hand, the operation is dealing with both negative and positive numbers, the result is invalid because the result is a negative number. In other words, two's complement *overflow* has occurred. Fortunately, the microprocessor will set a flag bit in a special register to warn the programmer of this condition. Dependant on the interpretation, the necessary steps can be taken in the program to either deal with an overflow or ignore the flag bit altogether.

Decimal and binary conversion

Although computers are quite happy with binary, humans are not. Most of us find a page full of '1s' and '0's monotonous and virtually unintelligible. Unless every bit is studied with great care, mistakes will be all too frequent. For example, try and spot (quickly) which bit is different in the following versions:

Version A	1001 0010 1101 0101 1110 1101 1101 1010
Version B	1001 0010 1101 0101 1110 1001 1101 1010

Fortunately, it is no great problem for software specialists to write binary to decimal conversion routines so that numbers can be entered

in normal decimal form. Indeed, when writing programs in high level languages such as BASIC or PASCAL, it may not even be necessary to understand binary since all entries and transactions take place via decimal numbers. It would seem that similar routines for shielding us from the boredom of binary could be employed in machine code programs. Indeed, most machine code assemblers allow decimal entries. But there is a snag because, even with an assembler, it is often necessary to be aware of the binary bit pattern corresponding to a decimal entry. Unfortunately, decimal and binary are far apart in the number stakes and it is by no means easy to visualise the binary pattern corresponding to the decimal equivalent or vice versa. That is to say, few people could take a quick glance at the pattern 0110 0111 and recognise the decimal equivalent. The problem is even worse for double byte patterns. What is needed is a numbering system which is a compromise between decimal and binary. A system which will enable a human to recognise almost instantly the associated binary pattern. One numbering system, popular in the earlier computers, was *octal* but this was eventually superseded by the now popular *hexadecimal* system.

Hexadecimal

The *base* of a numbering system is the number of *different characters* used. Binary only uses the character 0 and 1 so the base is 2. Decimal uses the characters 0, 1, 2 . . . 9 so the base is 10. The hexadecimal numbering system (known simply as hex) has a base of 16 because it uses the characters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Notice that decimal and hex are the same for numbers up to 9 but the remaining six characters are the letters A to F inclusive. The table overleaf shows the relationship between the three numbering systems.

Machine addresses

When working in BASIC, we seldom need to concern ourselves with actual machine addresses, except of course when we use POKE or PEEK. In machine code, actual machine addresses are constantly in use and, here again, we shall find that hex is far better than decimal for specifying addresses. The Amstrad machine addresses, expressed in decimal, would be address 0 to address 65,535. Expressed in hex, the address range is from 0000 to FFFF which is tidier in appearance. Besides, it is easier to break down the address range into 'pages' each of 256 bytes. The two left hand digits can then represent the page and the two right hand digits can represent the address on that page. For example, the hex address A324 can be visualised as address 24 on page A3.

Binary	*Hex*	*Decimal*
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Note that one hex digit can represent any 4-bit pattern whereas two decimal digits are required for the patterns 1010 to 1111 inclusive. This means we can express the contents of a byte with only two hex digits or a double byte with only four. It should not take too much practice on your part before you are able to convert binary to hex almost at sight and with no hesitation. Here are some examples:

Binary	1001	0001	1111	1111	0111	1010
Hex	9	1	F	F	7	A

As a final impressive example, consider the pattern, 0111 0000 1111 1010. You should soon be able to convert this at sight to 70FA hex. How long would it take you to convert it to decimal? The Z80 instruction codes are far better written in hex than in decimal. In fact most assemblers, including the one we shall be using, will display them in hex code so this is another reason why we should make some effort to master hex.

Converting hex to decimal

Although anyone intending to take up machine code seriously should learn to think in hex, there are times when it is necessary to convert from hex back to decimal. You will find this a bit awkward at first because the place weightings of hex digits go up in powers of 16 instead of in powers of 10. Instead of progressing in units of 1, 10, 100, 1000 etc, hex progresses in units of 1, 16, 256, 4096 etc. (Note that 256 is 16 squared and 4096 is 16 cubed). Fortunately, we seldom

have to deal with hex numbers of more than four digits. Here are some examples of hex to decimal conversion:

3F hex=(3×16)+15=63 decimal
A2 hex=(10×16)+2=162 decimal
FF hex=(15×16)+15=255 decimal
101 hex=(1×256)+1=257 decimal
100A hex=(1×4096)+10=4106 decimal
2ACB hex=(2×4096)+(10×256)+(12×16)+11=10955 decimal
FFFF=(15×4096)+(15×256)+(15×16)+15=65,535.

You can of course use the Amstrad in direct mode for hex to decimal conversion.

Adding two hex numbers

We don't often want to perform actual arithmetic using hex numbers but it is well to underlying mechanism. Just remember that if the sum of any column exceeds F, a carry is passed onto the next column. Here are some examples:

	0F	FF	3E	F0FA
	01	01	56	0006
Sum	10	100	94	F100

Largest hex numbers

We listed earlier the numerical limits of bit strings expressed in decimal.

It is useful to also know these limits in hex. In the case of unsigned binary, the largest hex number is FF for single byte and FFFF for double byte.

In the case of two's complement, the largest single byte positive number is 7F (+127 decimal) and the largest negative is 80 (−128 decimal). For double byte two's complement numbers, the limits are 7FFF (+32767 decimal) and 8000 (−32768 decimal).

Binary coded decimal

There is another numbering system in use, known as binary coded decimal or simply, BCD. It is a system designed to bridge the gap between binary and decimal but, unlike hex, it does not use all of the possible binary patterns in a byte. Here is a table of binary to BCD conversions:

Binary	*BCD*
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	Illegal
1011	Illegal
1100	Illegal
1101	Illegal
1110	Illegal
1111	illegal

Here are some examples of BCD patterns:

0100 1000=78 decimal. 1001 1001=99 decimal.
0000 0101=5 decimal. 0110 0011=63 decimal.

Note that the two nybbles within a byte must be treated independently of each other and must be read as two four-bit patterns.

Adding two numbers in BCD format
Certain difficulties arise when we try to add two numbers in BCD format because of the possibility of creating one of the six illegal combinations. Study the following two examples of addition in BCD:

Add 5 to 23	0000 0101	(5)
	0010 0011	(23)
Sum	0010 1000	(28)
This is a valid result		

Add 8 to 23	0000 1000	(8)
	0010 0011	(23)
Sum	0010 1011	Illegal

The low order nybble of the result has fallen into the illegal band. The solution is, at first sight, rather weird. The addition is performed and, if the low order byte result is illegal, just *add a further six*. Let us add a further six to the previous illegal result:

	0010 1011	Illegal
Add 6	0000 0110	
	0011 0001	(31)

Note that adding the extra six has resulted in a carry from the low nybble to the high nybble. This procedure always works and can be summed up as follows:

1 Perform normal binary addition.
2 Test the result for illegalities (any nybble greater than 1001).
3 If illegal, add six.

The reason why this works is because the extra six cause the result to skip over the six illegal patterns, 1010 to 1111.

What use is BCD?

Unless your interests extend to linking external digital instruments to the Amstrad, the answer is, not much! It is becoming almost commonplace nowadays to provide digital instrumentation with sockets which can be interfaced to a computer. These readings are normally given in BCD format and may be designed to link up with a special input/output bus system, pioneered by Hewlett Packard, and now known as the IEE 74 standard. Such a bus is not provided on the Amstrad but no doubt its popularity may encourage manufacturers of add-ons to provide a suitable interface.

Logical operations

There will be time when it is necessary to alter, or perhaps examine, one or more particular bits within a byte without disturbing the remaining bits. For example, we may need to ensure that bit 4 is a '1' *without disturbing the remaining bits.* On the other hand, we may wish to ensure that bit 7 is a '0'. We may even wish to change the state of certain bits. Normal arithmetic is not of much use to us in these circumstances. Fortunately, most computers (including the Amstrad) are able to carry out three, so called *logical* operations called AND, OR and XOR. They are used in conjunction with a mask word in which the bit pattern is chosen in accordance with certain rules.

To ensure certain bits within a byte are '0'
Use AND with a mask as follows:
'1's in the mask will leave corresponding bits alone, '0's will ensure the corresponding bits are '0'.

To ensure certain bits within a byte are '1'.
Use OR with a mask as follows:
'0' in the mask will leave corresponding bits alone, '1's will ensure the corresponding bits are '1'.

To ensure certain bits in a byte are changed (flipped)
Use XOR with a mask as follows:
'0's in the mask will leave corresponding bits alone, '1's will ensure they are changed.

Here are some examples:

1 To ensure bit 7 in a byte is 0, AND it with the mask 0111 1111, (7F hex).
2 To ensure bit 3 in a byte is 1, OR it with the mask 0000 1000, (08 hex).
3 To ensure that bit 5 in a byte is changed, XOR it with the mask 0010 0000, (20 hex).

As a final example, if a byte originally contained 1101 1110 and we ANDed it with mask 0111 1111, it would then contain 0101 1110. The three logical operations are purely bitwise in character. That is to say, each bit is treated quite independently of the others. There is no such thing as a carry from one bit to another as happens in arithmetic operations.

Summary

1 Analogue computers measure. Digital computers count.
2 Digital computer circuits are arrangements of electronic switches so they remain free from calibration errors.
3 The two digital states can be named either HIGH and LOW or 1 and 0.
4 A byte is 8 bits and a nybble is 4 bits.
5 The rightmost bit in a string of bit is the 1sb and the leftmost bits is the msb.
6 The bits in a byte are numbered 0 to 7 not 1 to 8.
7 Double byte numbers require a separate memory address for each byte.
8 A single byte can hold numbers up to 255 in unsigned binary but a double byte can hold up to 65,535.
9 The least significant byte in a double byte is called the low byte and the most significant the high byte.

10 Signed numbers are held in two's complement form.
11 In two's complement form, the msb is treated as the sign bit, 0 for positive and 1 for negative.
12 The highest positive single byte number is +127 and the highest negative is −128.
13 Flipping a bit means to change it from 0 to 1 or 1 to 0.
14 The two's complement of a number is the same number but of opposite sign.
15 In two's complement form, only bit 7 of the higher order byte of a two byte number is treated as the sign digit.
16 Whether a binary number is interpreted as unsigned or signed binary is a matter for the programmer, not the machine.
17 Hex notation can be preferable to decimal when working in machine code.
18 Hex uses a base of 16 and uses the characters A to F to cover the range from 10 to 15 respectively.
19 Any single byte binary pattern can be expressed by two hex digits.
20 The hex limit in unsigned binary is FF for single and FFFF for double bytes.
21 Single byte, two's complement hex limits are 7F for positive and 80 for negative.
22 Amstrad decimal address range is 0 to 65,535. The hex address range is 0000 to FFFF.
23 BCD addition can result in illegal combinations but they can be cleared by adding a further 6.
24 BCD is not in general use outside the instrumentation field.
25 AND is used to clear bits.
26 OR is used to set bits.
27 XOR is used to change bits.

4 Entering and running programs

The facilities of the Hisoft DEVPAC assembler, and the simple execution of machine code programs from BASIC, are described in this chapter to get the reader started.

A BASIC loader

It is possible to write, execute, debug, save and manipulate machine code programs by staying in BASIC and relying heavily on the commands CALL, PEEK and POKE. The machine code bytes (forming the program) can be entered as a series of numbers in DATA statements which, with the aid of a READ loop, can subsequently be POKED into successive locations in memory. Program 4.1, written in BASIC, is an example of a simple machine code loader. Using this program, you save yourself the expense of buying an assembler. But, you will soon discover that there are more things to life than money. The saving of a few pounds, representing a small fraction of the total cost of a computer, will turn out to be poor compensation for the hours and hours of frustration you will encounter when you try to develop your own programs with the sole aid of a BASIC loader. The Z80A microprocessor is blessed with a rich and complex set of over 600 instructions, far more complex than the 6502 with which some readers may already be familiar. Each one of these has its own unique 'operation code' which, in most cases is a pair of hex digits. These numbers have to be looked up for each instruction and copied down into DATA statements. Writing machine code programs, even with the aid of an assembler requires a degree of concentration and care. Without one, the prospect is little short of desolate. Unless the program is trivial, you may spend many hours, perhaps many weeks, trying to find bugs. Even when (if?) you finally manage a successful run, the appearance of the program in the form of DATA statement digits will, even to you, become so meaningless after a few days that attempts to introduce modifications would be out of the question. The operation codes are numbers, the addresses upon which the

codes operate are numbers, all of which leads to the most unfriendly of environments. Our advice to you is to get hold of an assembler as soon as you can. If you try and manage to scrape along without one there is a good chance that you will soon give up the idea of machine code altogether and that will mean you have wasted money on buying this book! However, to cater for readers who are willing to made do with POKE and PEEK techniques, all listings in this book will include columns containing the machine code hex digits so that they can be set into the BASIC loader (Program 4.1).

Program 4.1 Machine code loader

```
10 REM BASIC LOADER
20 REM FOR MACHINE CODE BYTES
30 INPUT"How many data bytes are there";number%
40 ADDRESS%=&7000
50 FOR N%=0 TO number%-1
60 READ BYTE$
70 POKE ADDRESS%+N%,VAL("&"+BYTE$)
80 NEXT
90 END
100 '
110 '
120 REM THE DATA BYTES BELOW ARE PURELY
130 REM FOR EXAMPLE PURPOSES
140 REM
150 DATA 3A,00,71,5F,3A,01,71,21
160 DATA 00,00,55,06,08,29,17,30
170 DATA 01,19,10,F9,22,02,71,C9
```

Using an assembler

Assemblers come under the heading of utility software. They are to machine code as interpreters or compilers are to high level languages. They all provide a friendly environment in which to write machine code. Meaningless numbers can be replaced by meangingful *letter groups*, and absolute addresses, which again are numbers, replaced by *labels* chosen for their mnemonic value. All these improvements allow the machine code programmer to concentrate more on logic flow instead of cluttering the mind with trivia.

As mentioned in an earlier chapter, we shall be using the official assembler chosen by Amsoft:

The HiSoft DEVPAC Assembler,
Disassembler Editor and Monitor (SOFT 116).

From now on, any reference to the 'assembler' must be taken to mean the above product.

There are two versions available, one for the CPC464 available on tape and a disc version for use on the CPC664/6128 or the CPC464

with external disc drive. In all fairness, we should point out that there are several other good assemblers on the market which are written for the Amstrad machines. The fact that we have concentrated on this one should not be taken as a sign that we consider the DEVPAC in any way superior. After all, the 'best' assembler is the one you know! Most of them offer, more or less, the same variety of options, even though the syntax or some of the symbols may vary a little. If you happen to have a different assembler, you will probably find little difficulty in translating our assembly format to suit yours.

The assembler comes complete with detailed documentation on its use. There are a variety of options but it would be waste of time and book space to repeat them all here. It is sufficient to go over the main options, relying on default conditions as much as possible. The assembler manual must be consulted when more detail is required. As in most utility software, there are some options which are used frequently, some occasionally and one or two hardly ever. When you are new to the assembler, it is not a good idea to worry too much about all the various options. In what follows, it can be taken for granted that the ENTER key must be struck to terminate all orders to the assembler.

Initially, the most important exercises to try out will be:

1 How to load and operate the assembler so that, when you have learned a few instruction mnemonics, you can get cracking on entering some simple source code.
2 How to list the source code and correct typing errors.
3 How to assemble the source code into object code and how to interpret the meaning of all the columns in the assembled version.
4 How to correct errors if the assembler reports any.
5 How to save and retrieve the source or object code on disc, or tape as the case may be.
6 How to execute the machine code.

Note that running the code should normally be the last exercise on the list. It is a risky business attempting to run (execute) code before saving it. If you are new to the game (or even if you are an old hand), NEVER attempt to run a machine code program unless you have saved the source code first. Remember earlier warnings regard-ing the malevolence of a naked microprocessor. It is ridiculously easy to initiate a system crash. Only risk a run first if you positively enjoy re-loading the assember from scratch and key bashing all your work in again.

Loading the assembler

To load and enter the assembler from cassette tape simply press the CRTL and small enter keys simultaneously. The assembler takes about 131 seconds to load. The disc version is loaded in two parts.

Begin by typing:

LOAD"GENA31"

The assembler loader, written in BASIC, is now in memory ready to load the full assembler, so we can now type:

RUN

The HiSoft logo will appear on the screen, and the following prompt appears:

Load address?

This refers to the first address of the block where the assembler is to be loaded. A good address is 1000 decimal which is chosen as the default address by just pressing the ENTER key.

The next prompt is:

Load MONA now?

MONA is a machine code monitor and entirely separate from the assembler. We are not yet ready for this facility so just answer:

N

The screen now tells us that GENA31, the body of the assembler, is being loaded from disc. This takes a second or two from disc.

Using the assembler

After loading is complete, (the list of assembler options appears on the screen, followed by the prompt '>', indicating the assembler is waiting for orders. The assembly options are spelt out in full but you need only key the characters which appear in upper case.

Line numbers

Each source code instruction is preceded by a line number followed by a space. Note that multistatement lines separated by colons are not accepted in the assembler format as they are in BASIC. Line numbers can be entered manually but if you are simply typing in listings from your notes, books or magazines then the use of the 'Insert' assembler command will save a lot of time. This automatically feeds you with line numbers at a fixed increment. For example, if you want the first line number to be 100 and subsequent lines to be 10 apart, you would key:

I 100,10

The first line number, 100, will now appear at the left ready for you to begin entering the first source code instruction. On completion of each line, the next line number automatically appears. When you have finished entering code, press CTRL C to return back to the assembler editor.

Listing the source code
When you have reached the last line (usually the instruction RET), you will probably want to list your source code by typing L. This assembler command also formats the display fields on the screen. Any obvious typing errors at this stage can be edited. Assembler editing procedures are virtually the same as BASIC.

Once you are satisfied with the source code listing, the next stage is to assemble it. That is to say, convert the source code into object code.

Assembling the source code
To assemble the source code into object code that can be executed, type:

A

The screen will first respond with:

Table size:

The assembler needs to know how much memory to reserve for the symbol table. In most cases it is sufficient to rely on the default setting by keying ENTER. If an error message 'No Table space' should appear then repeat with a higher estimated value. If all is OK then the screen responds with:

Options:

There are various assembly options available, including fast assembly with suppressed screen display and full format output to printer. You can ignore the options and rely on default by pressing ENTER which will bring forth the full assembly listing together with the corresponding object code. Like most assemblers, it requires *two passes* through the source code before it can completely translate into object code. If it discovers errors during the first pass, whilst it is scanning the labels and symbolic addresses, it will not attempt the second pass until you have edited out the errors. The number of errors are displayed in two digit form:

Pass 1 errors nn

If there are no errors, the second pass continues on from the first without pause and the full assembly listing together with the object

code listing is presented to the screen. To prevent high speed scrolling, the listing stops at the end of each screen 'page' and continues only after a key (any key) is pressed.

Making sense of assembly output

Viewing the full assembled output for the first time can be unsettling until you begin to understand what all the columns mean. Program 4.2 is an example assembly listing which we shall use for descriptive purposes. The details of the program, which happens to perform 8 bit integer multiplication, are quite irrelevant because, at this stage, we are concerned only with the meanings of the columns.

Assembler directives

We must distinguish between direct orders to the microprocessor, called *operation codes* (op codes for short) and orders given to the assembler itself, called assembler *directives* or 'pseudo codes'. Before describing them, note the meaning which the assembler attaches to the following characters:

(#) is used as a prefix denoting the following number is in hex.
(%) is used as a prefix denoting that the following number is in binary.
(If a number is used without either of the above prefixes, it is assumed to be decimal.)
(;) signifies that what follows is pure comment and will not be assembled (similar to a REM statement in BASIC).
($) can be used anywhere in the source code to refer to the current value of the location counter.

The most common assembler directives include the following:

The **ORG** (ORiGin) directive

Used to inform the location counter where the first byte of the machine code program is stored. In general, we shall stick to the address #7000 so the source code line could read:

ORG #7000

It is possible to choose a bad origin which could overwrite important data. The assembler checks this and may issue the warning 'Bad ORG!'

The **ENT** directive.

This assembler directive, which has no default setting, is used to specify the execution address of the object code. It is only needed if the code is to be executed with the assembler editor 'R' command.

For example: **ENT #7000** will force the code to be executed from address #7000 if the assember command 'R' is entered. If your code is intended to be executed from BASIC then ENT is not necessary.

The **EQU** (EQUate) directive

Used to enable labels of your own choice to refer to *absolute* addresses. Labels, can be in either upper or lower case and can be of any length but only the *first six* will be recognised. Spaces are not allowed as characters within labels. It is good programming practice to assign labels for addresses at the head of the source code so that they can subsequently be used in the body of the program. It makes the coding easier to read, a factor which contributes to the overall power of any assembler. Although it means a few extra lines to write in the source code, it adds nothing to the size of the assembled object code. Here is how you would assign the label 'begin' to the address #7000:

begin: EQU #7000

At any time later in the program, you can use 'begin' instead of #7000. We should emphasise that the choice of label is yours but, clearly, you should choose labels that bear at least a rough resemblance to the role they are to play.

Program 4.2 Example assembly listing

```
Hisoft GENA3.1 Assembler. Page     1.
Pass 1 errors: 00

                     10 ;EXAMPLE ASSEMBLER LISTING
7000                 20 begin:  EQU  #7000
7100                 30 top:    EQU  begin+#100
7100                 40 number: EQU  top
7101                 50 mult:   EQU  top+1
7102                 60 prod:   EQU  top+2
7000                 70         ORG  begin
7000   3A0071        80         LD   A,(number)
7003   5F            90         LD   E,A
7004   3A0171       100         LD   A,(mult)
7007   210000       110         LD   HL,0
700A   55           120         LD   D,L
700B   0608         130         LD   B,8
700D   29           140 shift:  ADD  HL,HL
700E   17           150         RLA
700F   3001         160         JR   NC,over
7011   19           170         ADD  HL,DE
7012   10F9         180 over:   DJNZ shift
7014   220271       190         LD   (prod),HL
7017   C9           200         RET
Pass 2 errors: 00
Table used:     93  from   136
```

Reading from left to right

Column 1 (Location counter)
This gives the absolute addresses where the object code bytes are stored. The address shown here is where the *first object code byte* of the second column is stored.
Column 2 (Object code)
Contains the set of object code digits which the assembler has translated from your source code.
Column 3 repeats the line numbers which were used in the source code.
Column 4 is the label field. The term 'labels' should be taken to mean not only address labels, defined under EQU, but also to destination labels for jumps.
Column 5 represents the assembler directives and machine op codes in mnemonic form.
Column 6 represent the 'operands' which form the second part (if any) of the machine instructions and assembler directives.

Assembler options

Note that columns 3, 4, 5 and 6 will be identical to the original source code before it was assembled. Only column 2 represents the final object code. When you asked for assembler output by entering A, you will remember that you were prompted for which assembler option? We advised that you could ignore this by pressing ENTER. However, if you respond with option 4, you will obtain a restricted assembler output giving only the addresses and object code (columns 1 and 2). It is these columns which are important to readers who have not yet obtained an assembler. More about this later.

Making sense of Program 4.2

It should now be possible to attack the details of Program 4.2.
Line 10 is purely a remark, which explains why there is no corresponding object code in columns 1 and 2.
Line 29 assigns the label 'begin' to the address #7000. Note that all labels appearing in column 4 must be followed by a colon delimiter.
Line 30 deserves detailed treatment because it illustrates several important features of labelled addresses. It assigns the label 'top' to 'begin' + #100. In other words, it assigns 'top' to an address #7100 which is #100 bytes further on than #7000. This little dodge shows how we can make use of arithmetical expressions in assignments. It also shows how a new label can be assigned in terms of a previous label. The fact that 'top' was chosen to be #100 bytes away from 'begin' was to ensure that there was plenty of room for the machine

code bytes in between. When you begin coding a program, you may not always know exactly how many bytes it will occupy so a somewhat extravagant estimate, in this case #100 bytes, is sufficient. You may observe that, because the code occupies addresses #7000 to #7017 it would have been sufficient to allow an extra #18 bytes so we could have written, 'begin' + #18 instead of 'begin' + #100. All this does is close up the gap between the end of the actual instruction lines and the beginning of the labelled addresses. In fact it is unwise to close the gap initially because it leaves no room for any extra lines which may be needed in the light of running experience. So our advice is, until you are finally satisfied with your creation, to allow a healthy margin of address space between the end of the program and the beginning of the labelled data. If the above method is adopted then only one EQUate directive need be changed to close up the gap.

Lines 40, 50 and 60 assigns the labels 'number', 'mult' and 'prod' to addresses 'top' (#7100), 'top'+1 (#7101) and 'top'+2 (#7102) respectively.

Line 70 informs the assembler to locate the code, starting at 'begin' (which is, of course, #7000).

Lines 80 to 200 is the actual machine code program. We shall make no attempt to explain it at the moment except to point out the following:

1 *Line 160* is an example of a conditional jump, similar to IF THEN . . . GOTO in BASIC. The assembly instruction, JR NC, over, is a conditional jump to the line labelled 'over'. Note that we can't jump to line numbers like we do in BASIC.

2 *Line 180* contains the instruction DJNZ which is another conditional jump. (It stands for 'Decrement the B register and Jump if Non Zero'). The jump, if the condition is satisfied, will be to the line with 'shift' in the label field.

3 *Line 200* is the end of the program and RET (similar to RETURN in BASIC) transfers control back from wherever it was called, usually the assembler editor, machine code monitor or BASIC.

The object code

If you have an assembler, the information under this heading may only be of academic interest. If you haven't one, it is of vital importance. Column 2 shows the object code and column 1 shows the addresses (in rather a roundabout way) in which each byte of the code is stored. Each line of the object code is in the form of hex digits, representing one machine code instruction. Each pair of digits occupy one byte of memory. Note that some instructions will require one byte, some two bytes, some three bytes and a few require four bytes of storage space. It is particularly important for those without an assembler to associate each instruction byte with

the memory addresses you have chosen. In the example case, we are only interested in addresses between #7000 and #7017. That is to say, only addresses in which the machine code bytes of Program 4.2 are stored. Commencing with the first instruction, we note it contains three pairs of hex digits 3A, 00 and 71. So the address #7000 is holding 3A, #7001 is holding 00 and #7002 is holding 71. The second instruction is the single pair, 5F and this is held in address #7003. Perhaps you can see now why the addresses in column 1 progress in ragged, rather than simple sequential, progression. The assembler shows the address linkages in this fashion in order to improve the appearance of the layout but we should remember that bytes, forming a complete instruction, are held *one beneath the other in sequential memory locations*. For example, the instructions in memory, bearing in mind that hex is just a shorthand way of representing binary, can be visualised as follows:

Address	*Memory*	*Hex*
#7000	0011 1010	3A
#7001	0000 0000	00
#7002	0111 0001	71
#7003	0101 1111	5F
#7004	0011 0101	3A
#7005	0000 0001	01
#7006	0111 0001	71
#7007	0010 0001	21
#7008	0000 0000	00
#7009	0000 0000	00
#700A	0101 0101	55
etc	etc	

Using the BASIC loader

We are now in a position to return to the BASIC loader, which appeared in skeletal form, as Program 4.1. The example bytes used in the DATA statements were taken directly from the assembly listing of Program 4.2. When you write machine code programs, you will have to enter your own program bytes in place of the examples shown. After you have entered them, you must count up how many there are. To facilitate counting, it is a good plan to always use 8 data bytes per line. This makes it easier to count. The last line, of course may have less than 8 byte pairs. On running Program 4.1, you will be asked how many bytes there are (in the example, there were 24). After running, the bytes will be stored in address #7000 onwards. To satisfy yourself that they have been correctly placed into memory you can PEEK a few of the locations. For example:

PRINT PEEK(&7000) should, in the case of the example, cause the decimal number 58 to appear (3A hex), whereas **PRINT PEEK(&7017)** should display the last byte, 201 decimal (C9 hex).

Although it may be obvious to many readers, it is still worth pointing out that when Program 4.1 is RUN, the machine code has only been *stored* in memory locations – you haven't yet run the machine code program so don't expect any results. In fact, the technique of running machine code programs will not be discussed until after we have fully treated the various ways of storing the code by means of assembler options.

Saving and retrieving code

The assembler contains commands for saving and loading source code (the manual refers to source code files as text files). Also, the object code bytes, generated by the assembler, can be saved on tape or disc as a binary file. You could, of course, use BASIC to save the object code but the assembler method saves us the bother of looking up the start and finish addresses. A binary file produced in this way can be loaded from BASIC and executed in the normal way via a CALL statement. However, be sure that your routine ends with RET or the routine will not return to BASIC after execution.

Saving source code (command P)

The command is given in the following format:

P first line,last line,filename
Example: **P 10,200,Mult**

This stores source code, on disc or tape, as a text file named Mult which starts at line 10 and ends at line 200. Note, the file name must not be enclosed within quotes.

Loading source code (command G)

The command is given in the following format:

G,,filename
Example: **G,,Mult**

This loads a text file named Mult from tape or disc. Note that start and end lines do not have to be specified this time.

Before loading a new file, it is normally advisable to delete any existing source code in memory. If this is not done, the loaded file

will be appended to the existing file and the whole renumbered with an increment of one.

Saving object code (command O)

The command is given in the following format:
O,,filename
Example: **O,,Multip**

Deleting source code (command D)

The command is given in the following format:
D n,m

This deletes from memory (not from disc or tape) all source code line numbers from n to m.
For example, to delete Program 4.2:

D 10,200

Renumbering source code (command N)

The command is given in the following format:

N n,m

This renumbers such that the first line is n and the increments are m. Both n and m must be quoted because defaults are not accepted.

To obtain option page (command H)

Just key **H** (Help) to obtain full list of assembler options.

To print hard copy (command Z)

(a) If all that is wanted is a printout of the source code listing, the command Z can be chosen from the Help page. The format being:
Z n,m

Assuming the printer is switched on, the command outputs lines n to m of the source code. If n and m are absent, the entire source code is printed by default.
(b) On issuing the command A (Assemble), various options are presented. Option 8 directs assembly listing to the printer. The

assembler automatically pages hard copy. That is to say, if the code is too long to be accommodated on one page of standard fanfold, sufficient lines feeds are sent in order to leapfrog over the creases. The assembler also prints the *page numbers* at the top of every page printed.

It is probable that many readers who purchase one of the Amstrad machines may already be in possession of an EPSON printer and may find that all text is printed with double spacing between lines. There is a hardware solution to this but, unless you have some practical experience in this field, you would be advised to rely on a software solution. Program 4.3 is a short program, which once run, at the start of a session, will solve the double spacing problem. We have used EPSON printers during preparation of the listings in this book.

Program 4.3 Epson printer mod.

```
10 REM AMSTRAD/EPSON PATCH
20 PRINT#8,CHR$(27);"A";CHR$(6)
30 WIDTH 50
40 END
```

Executing object code programs (command R)

At last, we arrive at the subject of running the object code. Once the source code file is saved on tape or disc and assembled, the object code can be executed from the assembler editor by typing R. However, this is only possible if the ENT directive has been included in the original source code listing. In a lot of cases it may not always be as simple as this. Most machine code programs, including our example Program 4.2, may require data to be present in specific addresses. During the introduction to Program 4.2, we mentioned that it was capable of multiplying two numbers together but, clearly, it will only work if the two numbers to be multiplied are present in the allotted locations. If you refer back to the program, you will see, after a bit of detective work, that these locations are at addresses #7100 and #7101, so it would be quite useless attempting to run the program unless the two numbers were present in advance. Furthermore, the result of the multiplication, which is two bytes in length, will be left in 'prod' and 'prod+1', (the low byte in address #7102 and the high byte in #7103).

Like most other 'programs', Program 4.2 is a subroutine. Subroutines require other programs or sections of code to supply any required data (called parameter passing) before they are called up. As far as this book is concerned, most of the listings given are indeed subroutines, intended to be called up from either another machine code program or from BASIC. Although the assembler provides the option 'R' for executing the object code, in nearly all

cases, you will be calling up the code from one or other of the above sources.

The line in BASIC for calling on the machine code or for calling from within another machine code program is:

CALL <address>

For example, after BASIC has obtained the necessary parameters from the key board and POKED them in to the required locations, **CALL &7000** would call up (execute) any code located at address &7000.

Switching between BASIC and assembler

When developing source code, you will be in the assembler environment. If you want to execute the object code after assembly, you can nip into BASIC by choosing option B. Please remember to save your source code first!

To come back to the assembler from BASIC you use:

CALL <load address + 4>.

For example, if you loaded the assembler at &2000, the BASIC command to get back to your source code listing in the assembler (a warm start) would be **CALL &2004.**

Returning back from BASIC to the assembler with a deleted source code file is termed a cold start and is effected with an offset of two

CALL <load address + 2>.

For example, **CALL &2002** will enter the assembler from scratch with all source code cleared so be careful here.
NOTE: when in BASIC, the prefix & is used to signify hex numbers. When in the assembler, the prefix is #.

The assembler manual

We have only attempted to describe the main points of the assembler in order to help you get started. In most cases, we have been content with simple default options. The manual supplied with the assembler, which must of course be considered the overriding authority, should be consulted for information on the other more fancy options.

Finally, we should emphasise that no attempt has yet been made to explain the actual machine code instructions. There is no point in learning these until you know how to use the assembler. This

requires practice. We suggest that you start by keying in the source code of Program 4.2, and learning how to edit, list, assemble etc.

Summary

1 When developing a new machine code program, save the source code before you execute any object code.
2 With the disc version LOAD'GENA31' loads the first part of the assembler (the loader). The remainder is loaded by typing RUN.
3 To assemble, means to translate source code into object code.
4 The assembler requires two passes through the source code before it can produce object code.
5 The assembler uses # to signify hex rather than &.
6 The semicolon ; before a source code line signifies it is only comment.
7 The character $ refers to the current value of the location counter.
8 Assembler directives, called pseudo codes, are orders to the assembler and will not appear in object code.
9 Assembler directives include: ORG for defining the address where the first object code is to be located and EQU for assigning labels to addresses.
10 After assembly, using the default option, columns 1 and 2 represent the addresses and object code, columns 3, 4, 5 and 6 are the original source code.
11 If Option 4 is used, only the object code is displayed.
12 It is customary to reserve locations at the bottom end, below the last instruction address, for data.
13 When reserving locations for data, make sure they are well below the last instruction address.
14 Object code can be executed from the assembler editor, machine code monitor or BASIC.
15 BASIC can be entered by typing in the assembler command 'B'.
16 Re-entry into the assembler from BASIC is effected by Call <load address + 2> for a cold start and CALL <load address + 4> for a warm start.
17 See the assembler manual for options not covered.

The Z80 registers 5

No attempt is made here to decribe the awesome internal complexity of the Z80. For programming it is enough to consider those components which can be directly or indirectly controlled by the programmer. These components are called *registers*.

What is a register?

A considerable amount of machine code programming is concerned with transferring data bytes between memory addresses or between one part of the microprocessor and another. The Z80, in common with most other 8-bit microprocessors, carries out most of its operations by means, or with the aid, of its internal registers. The registers are only eight bits wide so are only capable of holding unsigned (absolute) numbers up to 255 decimal (FF hex). However, the internal hardware design of the Z80 allows certain pairs of these registers to act as *double length* (16 bit) registers.

Source and destination rules

Before treating the registers in detail, it is important that you understand the mechanism of inter-memory or inter-register transfers. First we must differentiate between the *source* and the *destination*. For example, when we load data from A into B, then A is the source and B is the destination. These terms are almost self explanatory but note the following universal rule which applies to all transfers.

> After a transfer has taken place, the source data remains undisturbed but the old data at the destination is overwritten by the new data.

This is worth an example. Suppose A contained 20 and B 30 before the transfer and then we load A into B. Afterwards, A will still

contain 20 but so will B. The original 30 in B would have been overwritten.

Registeres are similar to normal memory locations in RAM, in that they can hold a byte of data. The most obvious difference, of course, is that they are located inside the microprocessor. But they differ in other, less obvious respects. For example, data movements into and out of most registers take place at much higher speeds than similar movements between memory locations. Also, there are differences in status between the registers. All memory locations are equal but when we come to registers, we find that some of them are more equal than others. One of them, called the *accumulator*, is blessed with certain privileges not enjoyed by the others – it is virtually the prima donna of all the single length registers. Although most of them can be used to store data, some registers are equipped for handling specific tasks over and above that of simple storage. For example, some can be used in pairs for handling double byte data, one is dedicated to the control of a special memory arrangement known as the *stack* and one is called a register (but shouldn't be) because it is merely a collection of quite separate bits acting as indication of certain *status* conditions. They are known as *flag bits.* Finally, one of the registers, called the *program counter,* is 16 bits long and is virtually in charge of the entire program flow.

The Z80 registers

The above general description of registers, which would apply equally well to several well known families of microprocessor must now give way to a detailed study of those found in the Z80. The Z80 is rather well endowed with registers. In fact a bank of eight of them are duplicated so, in theory, we can pick either set or switch from one set to the other. For the most part, we shall assume the alternate register bank does not exist because it is used by the Amstrad operating system. To tamper with any of these registers would be to court disaster. Bear in mind therefore that Figure 5.1 shows only the Z80 registers which are safe to use in your programs.

The details of each individual register will include example instructions written in standard Z80 assembly code.

Points to remember

Although we have already mentioned some of the rules regarding standard assembly code notation, it is worth repeating the rules conerning the order in which operands are written so there should be no doubt as to the meaning of the many examples which follow.

1 The first operand after the operation code is the destination and the second is the source. As an example, consider the hypothetical instruction:

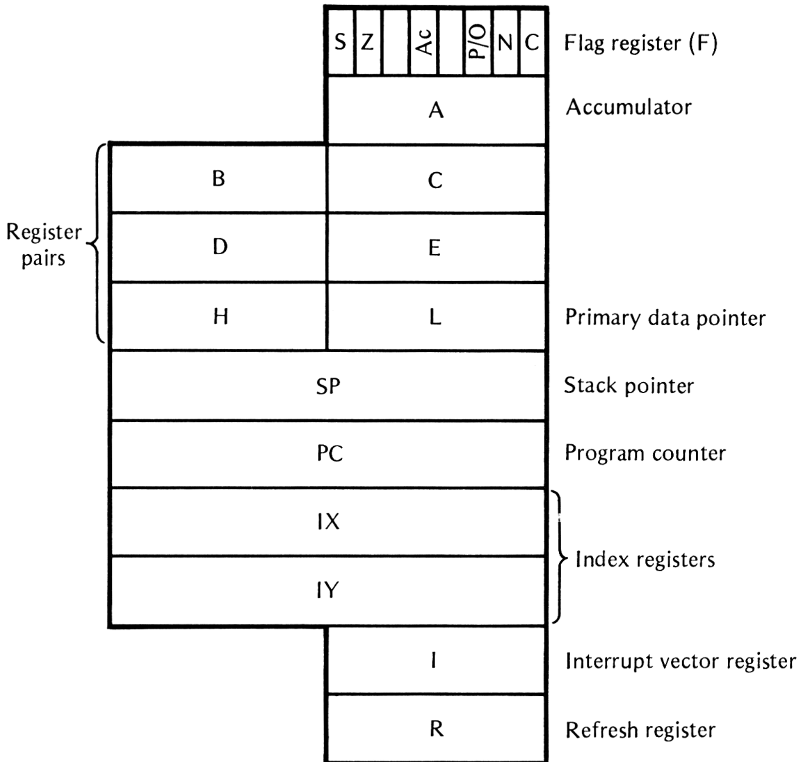


Fig 5.1 The Z80 registers

LD x,y

This will load y (the source) into x (the destination).
2 All numbers are assumed to be decimal unless prefixed by #, in which case they are interpreted as *hex* numbers.
3 Brackets round an operand should be taken as meaning 'The contents of'. For example, **LD A,20** means load the decimal number 20 into A. However, if we write **LD A, (20)** we mean load the contents of memory address 20 into A.

Now we are ready to examine the roles of the various registers within the Z80.

The accumulator (A)

As mentioned above, this is the primary register and has exceptional properties. These are:

(a) Data transfers between memory and accumulator take place faster than with other registers.

(b) There are more memory instructions for acting specifically on the accumulator than any other register.
(c) All 8-bit arithmetic and logical operations can only take place in the accumulator and the result will always be left in the accumulator. (The reference to logical operations, sometimes called Boolean operations, means AND, OR, XOR, etc which should already be familiar to BASIC users.)
To load 12 hex into the accumulator:

LD A,#12

To load contents of memory address 8034 hex into the accumulator:

LD A, (#8034)

To load contents of memory addressed by the register pair HL:

LD A, (HL)

Store accumulator in memory address 8034 hex:

LD (#8034),A

Add the number 12 hex to the accumulator:

ADD A,#12

Add the contents of register E to the accumulator:

ADD A,E

Add the contents of memory addressed by the register pair HL to the accumulator.

ADD A, (HL)

Single length registers B,C,D,E,H,L

These six registers can be used singly or in certain pairs for simple data interchange. The HL pair has special significance because of its specialised role as the *primary data pointer* for addressing memory, but more of that later. It is sufficient for the moment to treat the six registers as general purpose data locations.
Some examples follow.
To load register B from register E:

LD B,E

To load register E from register B:

LD E,B

In general, any one of the single registers A, B, C, D, E, H or L can be loaded into any one of the others, the general instruction being of the form:

LD destination,source

Register pairs

As shown in Figure 5.1, certain combinations of single length registers can be treated in pairs. The allowed register pairs are BC, DE and HL with the higher order byte always being held in the first named register of a pair. For example, in the pair BC, B will hold the high order and C the low order byte. Although there are no inter-register load instructions for any of the pairs, there is an instruction for exchanging data between DE and HL.
To exchange the data between DE and HL:

EX DE,HL

The primary data pointer (HL)

The HL register pair, in addition to its general purpose role, acts as a *memory address pointer* and is the recommended method of accessing memory. In fact, you are advised to reserve H and L solely for this purpose. The term address pointer means that the contents of HL is interpreted as the memory address of data, rather than the data itself. The general format for transferring data between any of the single length registers and memory is as follows:

LD register,(HL)
or **LD (HL),register**

Example 1. To load B with data at the address specified by HL:

LD B,(HL)

Example 2. To store A in memory at the address specified by HL:

LD (HL),A

It is, of course, necessary to ensure that HL already contains the required address before using it as an address pointer. One way of doing this would be the use of an immediate data load. For example,

to load A from address #6000 by indirect use of HL, we could use the following two instructions:

LD HL,#6000
LD A,(HL)

At first sight, this may seem a round about method – why not write **LD A,(#6000)** and save an instruction? However, it may be that some action is to be performed many times but acting on sequential memory addresses. The address pointer method allows the contents of HL to be changed (incremented or decremented) each time round the loop so that the *same instruction* can be made to act on sequential memory addresses. These advantages should be more apparent when loops are discussed in Chapter 10.

Although HL is preferable, the register pairs BC or DE may also be used as data pointers. For example, the above two instructions could have been written:

LD BC,#6000
LD A, (BC)

It is worth mentioning that using a register pair as an address pointer is referred to as *implied addressing* in Z80 literature. This can cause some confusion because the term is often used differently. For example, implied addressing in 6502 microprocessor literature is taken to cover those instructions in which a certain register (usually the accumulator) is implied rather than explicitly mentioned.

The index register pairs (IX and IY)

The two double length registers IX and IY are true 16 bit registers, similar in some respects to the HL address pointer in that their contents can be treated as a memory address. In fact all memory references using HL can alternatively be specified by using IX or IY. However, the index registers can have their contents modified by the addition of a constant so that the effective address pointer is (IX+constant) or (IY+constant). The following example:

LD A, (IX+#16)

will load into A the contents of memory at the address specified by adding #16 to IX. If IX contained #6000, then A would be loaded from the address #6016. Note that this is only a limited form of indexing. Normally, when computer engineers speak of indexed addressing, they expect the contents of an index register to be added to the operand. For example, an instruction like LD #6000,X where the contents of X is added to the operand #6000 to obtain the effective address. However, the Z80 has no provision for such true

indexing and must be considered a rather inferior imitation. In fact, there seems to be no apparent advantage is using **LD A,(IX+ constant)** over implied addressing using HL as the address pointer. A constant, by definition, can not be varied so it would appear to be of limited use in loop work.

The stack pointer (SP)

This is a true 16 bit register (not a register pair) specifically designed to act as an address pointer for accessing an area of memory known as the stack. It should never be used as a general purpose storage register. The stack acts as a Last In First Out column of memory locations (abreviated to LIFO). Any of the register pairs can be stored on the stack using **PUSH** or reclaimed from the stack by using **POP**. The use of these two instructions are described in Chapter 6.

Although the position of the stack in memory is not fixed by hardware, the Amstrad firmware routines initialise the stack pointer to immediately below #C000 so any attempt to alter its contents could turn out to be a somewhat hazardous exercise. There is no need to alter the stack pointer each time you want to use the stack because, as we shall see later, it is done automatically when **PUSH** or **POP** is used. Bear in mind that the stack 'grows downward' from the highest address as indicated above.

In practice, the Amstrad allows over 256 bytes of stack locations but it is extremely unlikely that all this will be used.

Use of the stack

In general, treat the stack as a temporary dumping ground. Situations often arise where a register, already containing important data from a previous operation, is required for other purposes. The register can be freed by temporarily dumping the data on the stack by the use of **PUSH** and later recovered by the use of **POP**. Note that the stack pointer, SP, is automatically decremented with each **PUSH** but incremented with each **POP**.

The program counter (PC)

Digital computers are said to be *sequence controlled*. That is to say, all instructions are executed automatically, one after the other, in strict address sequence. (There are some exceptions to this but for the moment, this is unimportant.) The 16 bit register, responsible for maintaining this sequence, is called the *program counter*, consequently, it is in complete control of a program. The contents of this register is the address in memory of the next instruction byte to be executed. The address of the first instruction byte in all programs must be set into the program counter. Armed with this information, no further action is required by the operator because all subsequent

instructions are fetched and executed automatically in strict address sequence. If the address of the first instruction to be executed (ignoring assembler directives) is at, say, #6000 the program will fetch this byte and execute it. The program counter is then automatically incremented by 1, ready for fetching the next instruction byte at address #6001.

Normally, each instruction byte of the program is fetched from the next sequential address because, as we have seen, the program counter is automatically incremented after each byte is executed. However, as mentioned above, there are exceptions to this rule. Certain instructions are able to force the program counter to depart from the normal rhythm by causing the program counter to be loaded with an entirely different address. This means that some instructions may be skipped over, or earlier instructions may be repeated. It is worth mentioning that this break in the rhythm is not carried out by a register load type instruction. Indeed, there are no equivalent LD instructions for acting on the program counter. Instead, the contents are changed as the result of executing 'jump' instructions. There is a rich variety of jump instructions, some unconditional and some conditional on the state of certain flags.

The flag register (F)

When conditional jumps are made, the criteria which decide whether or not the jump is to be executed, depends on the state of a certain bit, or bits, in the *flag register*. This register is not, in any general sense, a register at all. We can't store anything in it by programmed instructions. It is a collection of separate bits, each entirely independent of each other. It is part of the microprocessor control circuitry, keeping watch on the status of the machine. It does this by automatically setting or resetting various flag bits according to the result of certain instructions. Figure 5.2 shows how the bits are named.

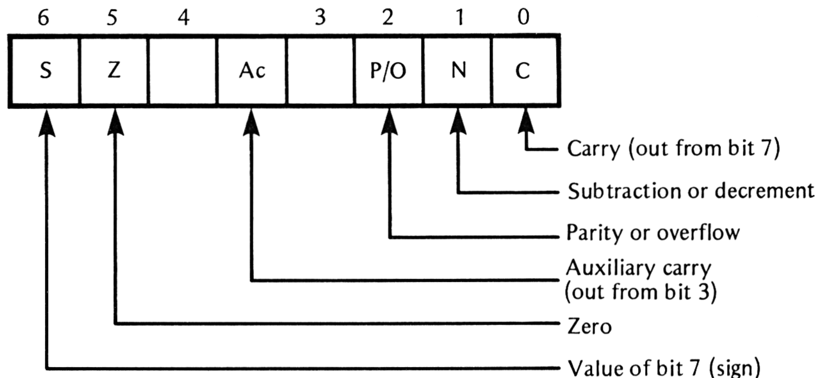


Fig 5.2 The flag register

Note that bits 4 and 6 are there but are not used. They will be permanently set to 1 or 0 so we can forget them. The flag bits have the following significance:

The C bit
This is at 1 if there has been a carry out from bit 7 of a register. If not, it will be 0.

The N bit
This is 1 after a subtract operation but 0 otherwise.

The P/O bit
This bit has two different meanings (Parity or Overflow) depending on the class of instruction.

After logical instructions, it is at 1 for even parity, 0 for odd parity (parity is discussed later).

After arithmetic operations, it will be 1 if there is two's complement overflow, otherwise it will be 0.

Ac bit
Ac means Auxiliary carry. It is 1 when a carry has passed between bit 3 to bit 4, otherwise it is 0. If you refer back to the paragraphs on BCD in Chapter 3, you will notice that the half carry passes from the right-hand to the left hand nibble following the result of an illegal combination.

The Z bit
Set to 1 if register contains zero, otherwise it is at 0.

The S bit
This is the sign bit for two's complement arithmetic. It is 1 if bit 7 of a result is 1, otherwise it is 0. Note that the term 'sign' bit is only relevant when applied to two's complement arithmetic. If the programmer is working in unsigned (absolute) numbers, the S bit has no significance. In such cases, it only indicates that bit 7 is set.

As mentioned above, not all instructions effect the flag register bits so it would be untrue to say that they always represent the conditions resulting from the 'last' instruction. For example, none of the load instructions having the mnemonic LD have any effect on the flag register but, as expected, all the arithmetic and logic instructions have an effect. As we shall see later, knowing whether or not a certain instruction affects particular bits within the flag register is of vital importance when programming conditional jumps. We must be careful not to assume that the flag bits are always determined by the results of the last instruction because not all instructions affect the flag register. In fact, the current status may have been the result of an earlier instruction. Although conditional jump instructions act on flag information they do not themselves have any effect on them.

The interrupt vector and the Refresh register

These control highly sensitive areas. Unless you are very experienced, treat them as unexploded bombs and keep well away.

Summary

1 Registers are high speed storage locations within the microprocessor.

2 The primary registers A, B, C, D, H and L are each 8 bits wide.

3 An alternative set of registers labelled A′, B′, C′ etc, duplicate the primary registers but should normally be left alone.

4 Registers B and C, D and E, H and L can be used in pairs, so acting as 16 bit registers.

5 The source register contents are preserved, the destination register contents are overwritten by the transferred data.

6 The first operand in assembly code is the destination, the second operand is the source.

7 Numbers are assumed decimal unless prefixed by #, in which case they are in hex.

8 The accumulator is register A and is privileged.

9 All 8 bit arithmetic and logical operations are carried out with the result in A.

10 Data can be exchanged between register pairs, HL and DE.

11 Any register pair can be used as a data pointer but HL is the fastest and recommended pair.

12 The IX and IY registers can be used for a limited form of indexing.

13 The stack is an area in memory. As data is entered, it grows downwards from the highest address.

14 The stack pointer register is the data pointer for the stack.

15 The program counter is the instruction pointer. Its contents being the address of the next instruction.

16 The program counter is incremented after each instruction byte, but after jump instructions, it could hold an abrupt out-of-sequence address.

17 The flag register is a collection of separate bits indicating certain conditions exist. Bits 3 and 5 are not used.

18 Flag register bits are not programmable. They are altered automatically by certain instructions.

19 Jump instructions depend on the current bits in the flag register but do not alter them.

Commonly used instructions 6

The Z80 has over 600 different instructions. To attempt a detailed description of them all could shatter the confidence of many readers new to machine code. Although the full list is set out in Appendix 1, it is better to concentrate on a subset of the most commonly used instructions, leaving the more exotic varieties till later. Even then, you may still find some of them will lie gathering dust unless, of course, you nurse an ambition to write your own operating system or BASIC interpreter.

Instruction mnemonics and operands

The assembler normally requires two items of information before it can 'understand' exactly what you want it to do. It must be told:

(a) WHAT particular action you want. This is called the instruction *mnemonic*.
(b) WHERE the data can be found. This is called the *operand*. The operand itself may often specify two registers, sometimes one register and a memory address and sometimes register pairs. In some cases, no operand is required because the instruction mnemonic itself is deemed to be sufficient.

Example 1: **LD A,B**

LD is the instruction mnemonic meaning LoaD. A and B together constitute two parts of the operand. The full meaning becomes:
Load the B register into the A register.
(Remember from Chapter 4 that the destination of the data is written first and the source last.)

Example 2: **ADD A, (HL)**

ADD is the instruction mnemonic meaning, in this case quite literally, ADD. The A register and (HL) are the two parts of the

operand. The full meaning becomes:

Add the contents of the memory location specified in the register pair HL to the accumulator contents.

Example 3: **RLA** and **RET**

RLA is the instruction mnemonic meaning Rotate Left the Accumulator through carry. RET means RETurn from subroutine. These two are examples of instructions which no not require an operand.

Addressing modes

There are a number of different ways of specifying the location of the data. They are known as *addressing modes*, and the following formal definitions should be studied. When entering instructions, take care with brackets, commas and spaces.

Implied addressing
The register pair (normally the HL pair) acts as a data pointer, holding the address of the desired memory location.

Example: **LD A, (HL)**

A few instructions allow the BC or DE pairs to be used as the data pointer. (Note: users with experience of the 6502 may be surprised at this definition of implied addressing.)

Direct addressing
The operand contains the direct memory address of the required data. Example **LD A, (#7100)** or **LD A, (label)** where 'label' is a previously assigned address.

Where a 16 bit load is required, such as in **LD HL, (#7100)**, the address quoted contains the low byte memory location which is loaded into L and the next address, #7101, contains the high byte which is loaded into H.

Immediate addressing
The operand contains the actual data. That is to say, it is 'immediately' available without going to memory.

Example: **LD A,#FF** or **LD HL,#FF34**

Program relative addressing
This only applies to conditional or unconditional jump type instructions. In object code, the operand indicates the number of program bytes to be 'jumped over', either forward or backward. The maximum number of bytes forward (counting from the first byte of

the instruction) is 129 and the maximum backwards is 126. Without an assembler, the programmer must count the number of bytes in order to find out what the operand number must be.

In assembly code, the operand is merely a label since the counting is done automatically by the assembler.

Example **JR FINISH** would be unconditional jump to the line which had FINISH in the label field so the programmer is unaware that program relative addressing is used.

Register indirect

Normally, the term indirect addressing is applied to an operand signifying a memory address which, in turn, contains the address of the required data. With register indirect, the operand specifies the register which contains the address.

Example: **JP (HL)** would jump to the address contained in the register pair HL.

Indexed addressing

The contents of an index register plus a 'displacement' value represents the memory address of the desired data. There are two index registers IX and IY.

Example: **ADD A, (IX+#15)**

If IX contained #FF61, the effective address would be #FF76. Either of the two 16 bit index registers, IX or IY, can be used. This is a rather limited form of indexed addressing because the displacement must be a constant, so it would appear to have little advantage over implied addressing. In fact, if the constant is zero, the action is identical to implied addressing except that IX or IY is used in place of HL as the data pointer. Nevertheless, indexed addressing provides a convenient way of passing parameters to a subroutine.

Instruction mnemonics and operation codes

An instruction mnemonic, such as **LD** or **ADC** makes sense only to an assembler. The equivalent, when using pure machine code, is called the *operation code*. Although there are over 600 different instructions, and therefore the same number of different operation codes, there are nowhere near the same number of different instruction mnemonics. This is because a number of different operands may share the same mnemonic.

This may be a relief for assembler users because it cuts down the apparent number of 'different' instructions. For those without one, the outlook is decidedly gloomy because each operand variant requires a unique instruction code.

The set of **LD** mnemonics provide an obvious example because, apart from loading register pairs to and from memory, it is possible to load any single register into any other. Since there are 7 single registers, the number of perms for these alone account for 49 different operands and therefore give rise to 49 different operation codes.

Presenting the instruction set

An instruction set which purports to cover every aspect of each instruction may be useful for experienced programmers but quite frightening for others. We shall adopt a middle course and restrict information to that considered essential for a first reading. Such information will be given under the following headings:
1 *The instruction as written in assembly code*
Example **LD A, (HL)**

2 *Operation code*
This will be given in hex digits or, in some cases, binary. It forms the first byte (in some cases the first two bytes) of the object code and may be followed by one or two operand bytes. They are of interest only to those without an assembler. The details of how to interpret and use the codes will be delayed until the end of this chapter.

3 *Meaning*
Given in plain English, rather than symbolic language. An example sometimes follows.

4 *Clock cycles*
This means how many effective Z80A clock cycles the instruction takes. The Z80A runs at 4 MHz but the *effective* frequency is about 3.3 MHz (refer back to Chapter 2). Thus, each clock cycle takes about 0.3 microseconds. Knowing the number of clock cycles each instruction takes may help in cases where high execution speed is essential and may help in choosing from several alternatives. (As in BASIC, there are various ways of achieving the same objective.)

5 *Bytes*
This is the number of bytes which each complete instruction occupies in memory after the source code has been assembled. Since each byte requires one memory location, this information may be important if memory is at a premium. It is also useful for those without an assembler because it provides information as to the number of operand bytes (addresses etc) which must follow the operation code. For example, if the operation code has 1 byte and the total number in the complete instruction is 3 bytes, then it follows that the operation code must be followed by 2 operand bytes.

6 Flags updated on result
Some instructions, particularly the arithmetic, logical, shift and rotate instructions, have an effect on certain bits in the F register (flag register). The three most important are the S, Z and C flags. For example, after an arithmetic instruction such as ADD, the S flag will be set to 1 if the result is negative, the Z flag will be set to 1 if the result is zero and the C flag set to 1 if the result causes a carry out at the msb end of the accumulator. Conditional jump instructions 'look' at the state of a particular flag to decide whether to jump or not to jump.

Where information under this heading is not given, it can be assumed that the instruction has no effect on the flags. To simplify the description of each instruction mnemonic and all operand variants, the abbreviations given in Table 6.1 will be used.

Table 6.1

Abrev.	*Meaning*
A,B,C,D,E,H,L,F	Any of the registers
pr	Any register pair BC,DE,HL,AF
rp	Any register pair BC,DE,HL,SP
reg	Any of the registers A,B,C,D,E,H,L
dst	destination register
src	source register
SP	Stack pointer
xy	Index register IX or IY
disp	8 bit signed binary displacement
addr	16 bit memory address
( )	contents of
IX	the X index register
IY	the Y index register
rr	(see end of Chapter)
rrr	(see end of Chapter)
ccc	(see end of Chapter)

Armed with these abbreviations, we proceed with the description of the commonly used instructions, although the choice of what constitues 'commonly used' will, to some extent, be governed by personal preference or habit. (The full instruction set is given in abbreviated form in Appendix 1 and Appendix 2.) We begin with the list of instruction mnemonics treated in this chapter.

ADC, ADD, AND, BIT, CALL, CP, DEC, DJNZ, EX, INC, JR, LD, POP, PUSH, RET, RLA, RRA, SUB, SBC, SLA, SRL, SUB.

We should mention that where absolute numerical addresses are quoted below, it is acceptable, (in fact advisable) to replace them with previously assigned labels.

Load instructions

LD dst,src
Operation code: 01rrrrrr
Meaning:
Load source register into destination register.
Example: **LD B,C** will load the C register into the B register. LD A,B will load the B register into the accumulator.
Clock cycles: 4
Number of bytes: 1

LD A,(addr)
Operation code: 3A
Meaning:
Load accumulator from memory using direct addressing.
Example: **LD A, (#7100)** load accumulator with contents of address #7100.
Clock cycles: 13
Number of bytes: 3

LD (addr),A
Operation code: 32
Meaning:
Store accumulator contents in memory, using direct addressing.
Example: **LD (#7100),A** stores accumulator in memory at address #7100.
Clock cycles: 13
Number of bytes: 3

LD reg, data
Operation code: 00rrr110
Meaning:
Load register with 8 bit data, using immediate addressing.
Example: **LD A,#B3** loads accumulator with #B3.
Clock cycles: 7
Number of bytes: 2

LD rp,data
Operation code: 00rr0001
Meaning:
Load register pair with 16 bit data using immediate addressing.
Example: **LD HL,#FF56** loads register pair HL with #FF56.
Clock cycles: 10
Bytes: 3

LD reg,(HL)
Operation code: 01rrr110
Meaning:
Load contents of memory, at the address specified by HL into

register using implied addressing.
Example: If HL contains #FF56, then **LD C,(HL)** will load register C with contents of address #FF56.
Clock cycles: 7
Bytes: 1

LD (HL),reg
Operation code: 01110rrr
Meaning:
Store contents of register in memory address, using implied addressing.
Example: If HL contains #7100, then **LD (HL),C** will store register C in address #7100.
Clock cycles: 7
Bytes: 1

LD (addr),HL
Operation code: 22
Meaning:
Store contents of HL using direct addressing.
Example: **LD (#7100),HL** will store L in address #7100 and H in address #7101.
Clock cycles: 16
Bytes: 3

LD (addr),rp
Operation code: ED 01rr0011
Note: this has a 2 byte operation code.
Meaning:
Store contents of register pair, using direct addressing.
Example: **LD (#7100),DE** will store E in address #7100 and D in address #7101.
Clock cycles: 20
Bytes: 4

LD HL,(addr)
Operation code: 2A
Meaning:
Store contents of HL, using direct addressing.
Example: **LD HL,** (#7100) will store the contents of address #7100 in L and the contents of #7101 in H.
Clock cycles: 16
Bytes: 3

LD rp,(addr)
Operation code: ED 01rr1011
Meaning:
Load register pair using direct addressing.

Clock cycles: 20
Bytes: 4

EX DE,HL
Operation code: EB
Meaning:
Swap contents of DE with HL
Clock cycles: 4
Bytes: 1

Arithmetic instructions

ADC A,data
Operation code: CE
Meaning:
Add the data, together with carry bit (if any), to the accumulator using immediate addressing.
Example: **ADC A,#03** will add #03 to the accumulator although, if the C bit is 1, it will add #04.
Clock cycles: 7
Bytes: 2
Flags updated on result: C,Z,S,Ac and P/0

ADD A,data
Operation code: C6
As **ADC A,data** except that carry bit is not involved in the actual addition.

ADC A,(HL)
Operation code: 8E
Meaning:
Add contents of memory, together with carry bit, to the accumulator, using implied addressing.
Clock cycles: 7
Bytes: 1
Flags updated on result: C,Z,S,Ac and P/0

ADD A,(HL)
Operation code: 86
As **ADC A, (HL)** except that carry bit is not involved in the actual addition.

ADC A,reg
Operation code: 10001rrr
Meaning:
Add contents of specified register, together with carry bit, to accumulator.
Clock cycles: 4

Bytes: 1
Flags updated on result: C,Z,S,Ac and P/0

ADD A, reg
Operation code: 10000rrr
As **ADC A,(HL)** except that carry bit is not involved in the actual addition.

ADC HL,rp
Operation code: ED 01rr1010
Note: this is a 2 byte operation code.
Meaning:
Add register pair, together with carry bit (if any), to HL.
Clock cycles: 15
Bytes: 2
Flags updated on result: C,Z,S

ADD HL,rp
Operation code: 00rr1001
Meaning:
Add register pair to HL.
Clock cycles: 11
Bytes: 1
Flags updated on result: C

SBC A,data
Operation code: DE
Meaning:
Subtract the data, and the carry bit (if any), from the accumulator using immediate addressing.
Example: **SBC A,#03** will subtract #03 from the accumulator although, if the C bit is 1, it will subtract #04.
Clock cycles: 7
Bytes: 2
Flags updated on result: C,Z,S,Ac and P/O. N is set to 1.

SUB data
Operation code: D6
As **SBC A,data** except that carry bit is not involved in the actual subtraction.

SBC A,(HL)
Operation code: 9E
Meaning:
Subtract contents of memory, and the carry bit, from the accumulator, using implied addressing.
Clock cycles: 7
Bytes: 1
Flags updated on result: C,Z,S,Ac, and P/O. N is set to 1.

SUB (HL)
Operation code: 96
As **SBCA, (HL)** except that the carry bit is not involved in the actual subtraction.

SBC A, reg
Operation code: 10011rrr
Meaning:
Subtract contents of specified register, and carry bit, from accumulator.
Clock cycles: 4
Bytes: 1
Flags updated on result: C,Z,S,Ac and P/O. N is set to 1.

SUB reg
Operation code: 10010rrr
As **SUB A,reg** except that carry bit is not involved in the actual subtraction.

SBC HL,rp
Operation code: ED 01rr0010
Note: this is a 2 byte operation code.
Meaning:
Subtract the designated register pair and the carry from HL.
Clock cycles: 15
Bytes: 2
Flags updated on result: C,Z and S. The N is set to 1.

Logical instructions

AND data
Operation code: E6
Meaning:
Performs the logical **AND** operation between the data, using immediate addressing, and the accumulator. Result is left in the accumulator.

Example:	Let A	= #35 (0011 0101)
	Let data	= #53 (0101 0011)
Afterwards, A		= #11 (0001 0001)

Clock cycles: 7
Bytes: 2
Flags updated on result: Z,S, and P/O but N and C are both reset to 0.

AND reg
Operation code: 10000rrr

Meaning:
Performs the logical **AND** operation between the data in the specified register and the accumulator. The result is left in the accumulator.
Clock cycles: 4
Bytes: 1
Flags updated on result: Z,S and P/O but N and C are both reset to 0.

AND (HL)
Operation code: A6
Meaning:
Perform the logical **AND** operation between the data, using implied addressing, and the accumulator. The result is left in the accumulator.
Clock cycles: 7
Bytes: 1
Flags updated on result: Z,S and P/O but N and C are both reset to 0.

OR data
Operation code: F6
Meaning:
Perform the logical **OR** operation between the data, using immediate addressing, and the accumulator. The result is left in the accumulator.
Clock cycles: 7
Bytes: 2
Flags updated on result: Z,S and P/O but N and C are both reset to 0.

OR reg
Operation code: 10110rrr
Meaning:
Performs the logical **OR** operation between the data in the specified register and the accumulator. The result is left in the accumulator.
Clock cycles: 4
Bytes: 1
Flags updated on result: Z,S and P/O but N and C are both reset to 0.

OR (HL)
Operation code: B6
Meaning:
Perform the logical **OR** operation between the data, using implied addressing, and the accumulator. The result is left in the accumulator.
Clock cycles: 7
Bytes: 1
Flags updated on result: Z,S and P/O but N and C are both reset to 0.

XOR data
Operation code: EE
Meaning:
Perform the logical **XOR** operation (eXclusive OR) between the data, using immediate addressing, and the accumulator. The result is left in the accumulator.
Clock cycles: 7
Bytes: 2
Flags updated on result: Z,S and P/O but N and C are both reset to 0.

XOR reg
Operation code: 10101rrr
Meaning:
Performs the logical **XOR** operation between the data in the specified register and the accumulator. The result is left in the accumulator.
Clock cycles: 4
Bytes: 1
Flags updated on result: Z,S and P/O but N and C are both reset to 0.

XOR (HL)
Operation code: AE
Meaning:
Perform the logical **XOR** operation between the data, using implied addressing, and the accumulator. The result is left in the accumulator.
Clock cycles: 7
Bytes: 1
Flags updated on result: Z,S and P/O but N and C are both reset to 0.

Increment and decrement instructions

INC reg
Operation code: 00rrr100
Meaning:
Increment the specified register by 1.
Clock cycles: 4
Bytes: 1
Flags updated on result: Z,S,Ac

INC rp
Operation code: 00rr0011
Meaning:
Increment the specified register pair by 1.
Example: **INC HL** will add 1 to the register pair HL.
Clock cycles: 6
Bytes: 1
Flags updated on result: None

INC (HL)
Operation code: 34
Meaning:
Increment the data in memory, using implied addressing, by 1.
Clock cycles: 11
Bytes: 1
Flags updated on result: Z,S,Ac

DEC reg
Operation code: 00rrr101
Meaning:
Decrement the specified register by 1.
Clock cycles: 4
Bytes: 1
Flags updated on result: Z,S,Ac but N is set to 1

DEC rp
Operation code: 00rr1011
Meaning:
Decrement the specified register pair by 1.
Clock cycles: 6
Bytes: 1
Flags updated on result: none

DEC (HL)
Operation code: 35
Meaning:
Decrement the data in memory, using implied addressing, by 1.
Clock cycles: 11
Bytes: 1
Flags updated on result: Z,S,Ac but N is set to 1

Stack operations

PUSH pr
Operation code: 11rr0101
Meaning:
Push the contents of the specified register pair on top of stack and decrement Stack Pointer by 2.
Clock cycles: 11
Bytes: 1

POP pr
Operation code: 11rr0001
Meaning:
Put the contents of top of stack into the specified register pair and increment Stack Pointer by 2.
Clock cycles: 10
Bytes: 1

Conditional jump instructions

JR condition, label
Operation code: see below.
Meaning:
There are 4 conditional jump instructions, using program relative addressing. In assembly code, if the condition is true, a jump is made to the line having that particular label in the label field. The maximum forward jump is +129 bytes, maximum negative is −126 bytes.
Clock cycles: 7 if condition is not met, 12 if condition met.
Bytes: 2

JR C,label
Operation code: 38
Meaning:
Jump relative if carry is set to 1.

JR NC,label
Operation code: 30
Meaning:
Jump relative if carry is reset to 0.

JR Z,label
Operation code: 28
Meaning:
Jump relative if result is zero.

JR NZ,label
Operation code: 20
Meaning:
Jump relative if result is non zero.

JP condition,label
Operation code: 11ccc010
Meaning:
There are 8 conditional jump instructions, using direct addressing which means that the jump address can be anywhere in memory. The particular condition is specified by one of the following letters:
C=jump if carry set to 1. ccc=011
NC=jump if carry reset to 0. ccc=010
Z=jump if result zero. ccc=001
NZ=jump if result non zero. ccc=000
PO=jump if P/O flag set to 0. ccc=100
PE=jump if P/O flag set to 1. ccc=101
P=jump if sign positive. ccc=110
M=jump is sign negative. ccc=111
Clock cycles: 10
Bytes: 3

DJNZ label
Operation code: 10
Meaning:
Decrement the B register by 1 and jump to the labelled line, only if the register is non zero. That is to say, the jump is made only if the Z flag is 0 after the decrement. Since relative addressing is used, the limits of the jump are forward 129 bytes and backwards, 126 bytes.
Clock cycles: 8 if condition not met, 13 if met.
Bytes: 2

Unconditional jumps

JP label
Operation code: C3
Meaning:
Unconditional jump to instruction which has that label in the label field.
Clock cycles: 10
Bytes: 3

JP (HL)
Operation code: E9
Meaning:
Unconditional jump to the address contained in HL.
Clock cycles: 4
Bytes: 1

Subroutine calls

CALL label
Operation code: CD
Meaning:
Jump to subroutine starting at address represented by label.
Clock cycles: 17
Bytes: 3

CALL condition,label
Operation code: 11ccc100
As above, except that the call is conditional.
Example: **CALL NZ,SORT** will call the subroutine at the address, represented by the label SORT, only if the Z bit is 0 (non zero result).
Clock cycles: 10 if condition is not met, 17 if met.
Bytes: 3

RET
Operation code: C9

Meaning:
Return from subroutine.
Clock cycles: 10
Bytes: 1

RET condition
Operation code: 11ccc000
Meaning:
As above except that the return is conditional.
Clock cycles: 5 if condition not met, 11 if met.
Bytes: 1

Comparison instructions

CP data
Operation code: FE
Meaning:
Compare data, using immediate addressing, with the accumulator and set flags accordingly. The comparison is made by subtracting the data from the accumulator. Original accumulator contents are restored after the comparison is made so the overall effect of the instruction is only on the flag register bits. It will normally be followed by a conditional jump instruction.
Clock cycles: 7
Bytes: 2
Flags updated on result: C,Z,S,Ac but N is set to 1

CP reg
Operation code: 10111rrr
Meaning:
As **CP data** above, except that the specified register is compared with the accumulator.
Clock cycles: 4
Bytes: 1
Flags updated on result: C,Z,S,Ac but N is set to 1

CP (HL)
Operation code: BE
Meaning:
As **CP data** above, except that the memory data, obtained by implied addressing, is compared with the accumultor.
Clock cycles: 7
Bytes: 1
Flags updated on result: C,Z,S,Ac but N is set to 1

Bit tests

BIT b,reg
Operation code: CB 01bbbrrr

Note: this is a 2 byte operation code.
Meaning:
The complement of specified bit (b) in specified register is placed in Z flag. The register contents are unaltered.
Example: Suppose register B contains 1011 0011, then **BIT 5,B** will make Z=0 (non zero). This is because bit 5 is a 1 and the complement of 1 is 0.
Clock cycles: 8
Bytes: 2
Flags updated on result: Only the Z flag.

BIT b,(HL)
Operation code: CB 01bbb110
Meaning:
As above except that bit b of the memory location addressed by HL, is tested instead of a register.
Clock cycles: 12
Bytes: 2

Shift and rotate instructions

(Refer to Figure 6.1 for illustration of instructions under this heading.)

RLA
Operation code: 17
Meaning:
Rotate accumulator left through carry.
Clock cycles: 4
Bytes: 1
Flags updated on result: C

RRA
Operation code: IF
Meaning:
Rotate accumulator right through carry.
Clock cycles: 4
Bytes: 1
Flags updated on result: C

SLA reg
Operation code: CB 00100rrr
Meaning:
Shift contents of specified register left arithmetic.
Clock cycles: 8
Bytes: 2
Flags updated on result: C,Z,S. The P/O flag reflects the party value.

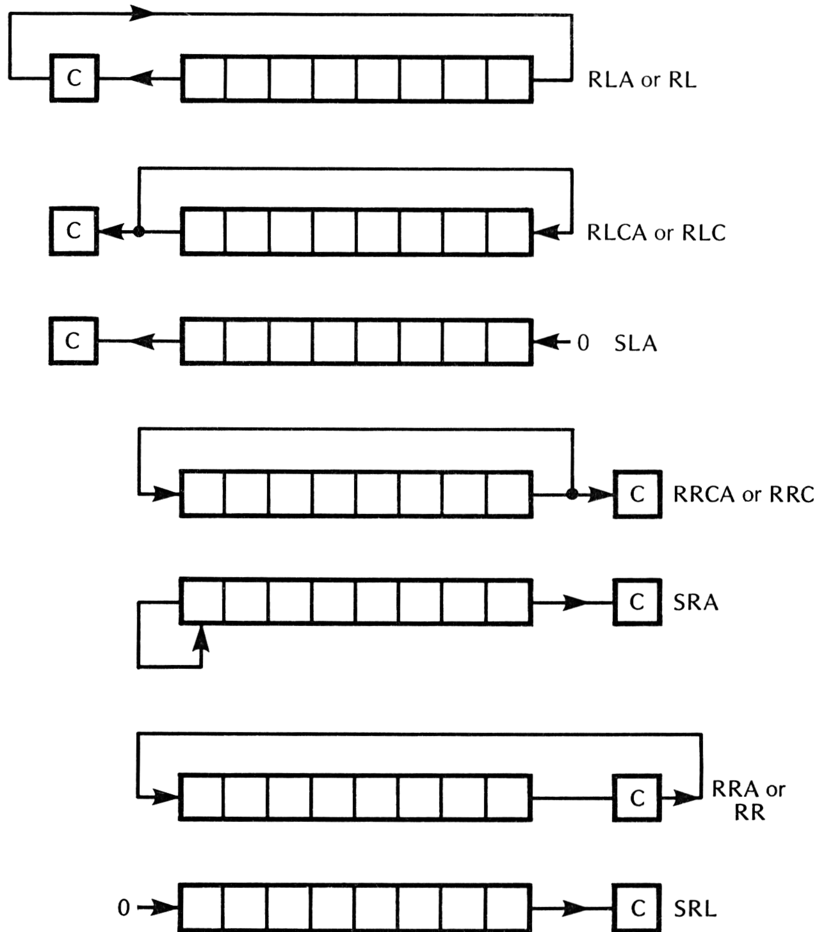


Fig 6.1 Shift and Rotate instructions

SLA (HL)
Operation code: CB 26
Note: this is a 2 byte code.
Meaning:
As **SLA reg**, except that the shift takes place on the data in memory, using implied addressing.
Clock cycles: 15
Bytes: 2
Flags updated on result: C,Z,S. The P/O flag reflects the parity value.

SRA (HL)
Operation code: CB 2E
Note: this is a 2 byte code

Meaning:
Arithmetic shift right contents of implied addressed memory location.
Clock cycles: 15
Bytes: 2

SRL reg
Operation code: CB 00111rrr
Meaning:
Shift contents of specified register right logical.
Clock cycles: 8
Bytes: 2
Flags updated on result: C,Z,S. The P/O flag reflects the parity value.

SRL (HL)
Operation code: CB 3E
Meaning:
As **SRL reg**, except that the shift takes place on the data in memory, using implied addressing.
Clock cycles: 15
Bytes: 2
Flags updated on result: C,Z,S. The P/O flag reflects the parity value.

Operation code details

As mentioned earlier, the details of the operation codes are primarily of interest to those who wish to plod on without the aid of an assembler. You will need the following information before you can understand operation codes which have been given in binary instead of hex digits:

1. *What rrr means:*

Register	rrr
A	111
B	000
C	001
D	010
E	011
H	100
L	101

2 *What rr means:*

Register pair	rr
BC	00
DE	01
HL	10
SP	11

3 *What ccc means:*

Condition	ccc
non-zero	000
zero	001
no carry	010
carry	011
odd parity	100
even parity	101
positive	110
negative	111

4 *What bbb means:*

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

We will start by comparing a few lines of object code with their equivalent in assembly source code in hex. The lines have been chosen for illustration only and are not intended to make sense as a program. Where the object code contains more than one byte, we have separated them by a space for clarity reasons.

Line	*Obj code*	*Label*	*Source code*
1	3A 15		**LD A,#15**
2	5F	BACK:	**LD E,A**
3	55		**LD D,L**
4	19		**ADD HL,DE**
5	17		**RLA**
6	20 03		**JR NZ,DOWN**
7	3A 43 71		**LD A,(#7143)**
8	ED 53 F7 7D	DOWN:	**LD (#7DF7),DE**
9	28 F1		**JR Z,BACK**
10	C9		**RET**

The first byte of object code in each line is the operation code, except for line 8 which has a 2 byte code. Any remaining bytes belong to the operand.

Line 1: this is easy because the instruction LD A,data, gives the operation code directly in hex. The operand is the immediate data 15.

Line 2: not so easy because the operation code for the instruction LD dst,src is given as 01rrrrrr. The destination register is E (code 011 above) and the source register is A (code 111 above). So the full code in binary is 01 011 111 which in hex is 5F. Note the line has BACK in the label field so it will be a jump destination.
Line 3: is another LD dst,src instruction, using L (code 101) as the source register and D (code 010) as the destination register. The binary code is therefore 01 010 101 is 55.
Line 4: is an example of the register pair add instruction ADD HL,rp. The operation code is given as 00rr1001 so from the table above, DE is 01. The full code becomes 00 01 1001 which is 19 in hex.
Line 5: another easy one because the code for RLA is given directly in hex.
Line 6: the operation code for JR NZ, label is 20. But why the operand is 03 will be explained soon.
Line 7: the instruction loads the contents of a 2 byte address into the accumulator. The operation code is 3A. The next two bytes represent the hex address 7143 but note carefully that the two bytes must be written back to front – low byte first.
Line 8: this instruction is one of the awkward ones because it has a two byte operation code ED 53. The operation code was given as ED 01rr0011. The register pair is DE so rr is 01. The full code is therefore ED 01010011 which is ED 53 hex. The next two bytes represent the address and, as always, must be written low byte first. Note the line has the destination label 'DOWN'.
Line 9: the operation code for JR Z,label is 28 but why the operand is F1 will be explained soon.
Line 10: operation code for RET is C9 and does not require an operand.

Entering the bytes

If this was a practical program and you wanted to enter it by means of the machine code loader, Program 4.1, the DATA statements would be laid out as follows:

```
150 DATA 3A,15,5F,55,19,17,20,03
160 DATA 3A,43,71,ED,53,F7,7D,28
170 DATA F1,C9
```

Counting relative jump bytes

We must now treat the mysterious operands of lines in lines 6 and 9 above. Relative addressing, as far as hex object code bytes are concerned, means the operand is a number, representing the number of bytes which must be skipped over in order to reach the labelled instruction. In other words, what number must be added to

the contents of the Program Counter in order to make the program jump back or forward? Now, in all instructions, including relative jump instructions, the program counter contains the address of the NEXT instruction byte. So we always begin the count by taking the next instruction byte as the reference point. To help in understanding line 6, the lines involved are repeated below:

```
6   20 03                   JR NZ,DOWN
7   3A 43 71                LD A,(#7143)
8   ED 53 F7 7D    DOWN:    LD (#7DF7),DE
```

The reference point is the byte 3A (the first one after the jump instruction). The destination is the byte ED so 3 bytes have to be added to the Program Counter. This explains why the operand in line 6 is 03.

Backward jumps provide an added difficulty because they count as 'negative' jumps. Thus, if we have to jump, say, 5 bytes backwards from reference, the operand will be the two's complement of 5 which is FB. Line 9 is a jump back to the labelled line 'BACK'. To understand why the operand is F1, the relevant lines are repeated:

```
 2  5F             BACK:    LD E,A
 3  55                      LD D,L
 4  19                      ADD HL,DE
 5  17                      RLA
 6  20 03                   JR NZ,DOWN
 7  3A 43 71                LD A,(#7143)
 8  ED 53 F7 7D    DOWN:    LD (#7DF7),DE
 9  28 F1                   JR Z,BACK
10  C9                      RET
```

The reference point is the byte C9, the first byte in line 10. The destination is the byte 5F in line 2. The jump is 15 bytes back and the two's complement of 15 is F1.

Dangers

It is easy to be one out in the count so be careful because the destination might end up on an operand, instead of an operation code byte. The microprocessor has no initiative and certainly no love for humans. It will gleefully seize any opportunity of crashing the system. For this reason, you are advised to save your object code before attempting to execute it.

Jumps using direct addressing

Jump instructions having the mnemonic JP cause no difficulty in coding because the operand is a straightforward 2 byte address in the form low byte, high byte. If you have no assembler, then this

addressing method will be the easiest, at least during the initial learning phase.

Summary

1 The instruction mnemonic determines the kind of action. The format of the operand determines the addressing mode.
2 A register pair holding the address of data is called an address or data pointer.
3 HL is the recommended register pair for use as an address pointer.
4 Implied addressing uses an address pointer.
5 Direct addressing uses the actual address (or pre-assigned label).
6 Immediate addressing doesn't use an address at all! The operand is the data.
7 Program relative addressing uses the operand as a signed displacement number to be added to the program counter.
8 The assembly language programmer is shielded from program relative addressing because a jump can be made to a labelled line.
9 Only a limited form of Indexed addressing is available using the IX and IY registers.
10 Execution speed of an instruction is determined by the number of clock cycles.
11 Memory locations occupied depend on the number of instruction bytes.
12 Arithmetic, logical, comparison, shift and rotate instructions update the flags.
13 Straightforward load, jump and stack instructions have no effect on status flags.
14 DJNZ only operates in conjunction with the B register.
15 The BIT instruction, in conjunction with the Z flag, can be used to examine the state of any bit in a register or memory location.
16 Shift instructions can lose bits at one end, rotate instructions cannot.
17 Operation codes are of vital interest only if instructions are to be POKEd from BASIC. That is to say without the aid of an assembler.
18 The first byte in a line of object code is the operation code. (A few instructions require 2 bytes). Any bytes which follow form the operand, usually an address.
19 Two byte operands are written with the low byte first.
20 In Relative addressing, backward branching (JR jumps) requires a two's complement operand.

7 Using resident firmware

Free software

The operating system in the Amstrad CPC464/664/6128 is a collection of machine code subroutines buried in the lower half of ROM. It occupies the block of addresses #0000 to #3FFF. We should consider these subroutines as a software goldmine, providing a rich source of ready made, rigidly tested machine code building blocks, many of which can be spliced into our own programs. To exploit any particular one, we need to know:

(a) Exactly what it does.
(b) The calling address.
(c) Preparation before entry.
(d) Where results, if any, are placed.
(e) The state in which registers and flags are left after the call.

Some readers, blessed with independent characters, might be vaguely disturbed at the thought of resorting to off-the-shelf subroutines. They could well argue that, taken to excess, the end result could be little more than high level language in disguise. There is, of course, an element of truth in this argument. However, if you are in the learning stage and are determined to go it alone by avoiding resident software altogether then all we can do is to wish you the best of luck. The apparently simple act of keying in a single numerical digit to the accumulator and echoing it to the screen could turn out to be far more difficult than you ever imagined. When (if?) you master this, then you face the even more difficult problem of entering and displaying a multi-digit number. The truth is that many everyday operations such as these and which we all take for granted in high level language, require a great deal of programming skill and patience before they become foolproof and flexible. It is one thing to get a subroutine to work but quite another to make sure it is foolproof under all conditions. Because of these obstacles, you should not hesitate to use resident subroutines for mundane tasks, at least until you achieve confidence and expertise. Perhaps, as your experience grows, you may reach the stage where you think you can

write them more efficiently yourself. If you have such ambitions, remember that you will be placing yourself in competition with professional operating system programmers.

There is a happy mean in everything. Rely too much on resident subroutines and you could end up with a program which is virtually a string of software beads written by someone else. On the other hand, trying to avoid them altogether merely to satisfy a craving for independence could lead to bouts of bad language and, in some cases, a vicious attack on the Amstrad.

Jump blocks

Although the subroutines are actually located in the addresses #0000 to #3FFF, they are not called directly from there. Instead, they are called from certain locations in RAM known as *jump blocks*. When the computer is first switched on, or following a *hard reset*, the operating system in ROM copies a series of jump instructions into these locations. The main jump block, the one in which we are interested, occupies addresses #BB00 to #BD39. Each element occupies *three bytes* so the subroutine call addresses in the jump block are always spaced three bytes apart. For example, the subroutine called READ CHAR is called at address #BB09. We shall not delve into the mysteries of how such a call is eventually routed to the actual subroutine in ROM. Those wishing to pursue the matter can easily invoke MONA3, the Disassembler, which is included in the HiSoft DEVPAC package. Don't worry about it though. There is no need whatever to understand it all. Just **CALL #BB09** and leave the rest to the operating system.

A few of the more commonly used ROM subroutines are given below. For the most part, they are concerned with simple keyboard and screen activities. For the complete list and more detailed specifications, you should consult the official Amstrad publication: 'Soft 158-Operating system Firmware Specification.

KM RESET (CALL #BB03)
Action: clears the key buffer and re-initialises the Key Manager indirections and buffers.
AF, BC, DE and HL registers are corrupted.

KM WAIT CHAR (CALL #BB06)
Action: waits until a character is available from the key buffer and loads its ASCII code into the accumulator.
Carry is always set but all other flags corrupted. Other register contents are preserved.

KM READ CHAR (CALL #BB09)
Action: similar to KM WAIT CHAR but does not wait. On exit the accumulator contains the character code. If character was available,

the carry is set. If not available, the carry is reset and accumulator corrupted.
After each call other registers are preserved but flags are corrupted..

KM WAIT KEY (CALL #BB18)
Action: waits, if necessary, for the next key from the key buffer and, after consulting a key translation table, loads either the character code or expansion token into the accumulator.
Carry is set. All other flags corrupted.
Contents of other registers preserved.

KM READ KEY (CALL #BB1B)
Action: similar to KM WAIT KEY but does not wait. On exit, if carry is set then a key was pressed and the accumulator contains the character code or expansion token. If the carry is clear then a key was not pressed and the accumulator is corrupt.
After each call other registers are preserved but flags corrupted.

TXT OUTPUT (CALL #BB5A)
Action: sends the character or control code in the accumulator to the screen or current stream.
No effect on registers and flags.

TXT VDU ENABLE (CALL #BB54)
Action: allows characters to be printed on the screen.
A and F registers only are left corrupted.

TXT VDU DISABLE (CALL #BB57)
Action: prevent characters being printed on the screen and disable cursor blob.
A and F registers only are left corrupted.

TXT RD CHAR (CALL #BB60)
Action: reads a character from the screen (at the cursor position of the currently selected stream) into the accumulator.
Carry is set if recognisable character is found. If not found, carry is reset and accumulator is cleared to zero.
After each call other register contents are preserved but flags corrupted.

TXT CUR ON (CALL #BB81)
Action: allow cursor to be displayed on screen unless it is disabled.
No effect on flags or registers.

TXT CUR OFF (CALL #BB84)
Action: prevent cursor from reaching screen.
No effect on flags or registers.

TXT SET PEN (CALL BB90)
Action: sets the text pen ink for writing foreground characters. The ink number must be in the accumulator before calling.
AF and HL register pairs are left corrupted.

TXT SET PAPER (CALL #BB96)
Action: sets the text paper ink (background colour). The ink number must be in the accumulator before calling.
AF and HL register pairs are left corrupted.

TXT INVERSE (CALL #BB9C)
Action: swaps the pen and paper inks over.
AF and HL register pairs are left corrupted.

GRA INITIALISE (CALL #BBBA)
Action: initialises the Graphics VDU.
AF, BC, DE and HL register pairs are left corrupted.

GRA MOVE ABSOLUTE (CALL #BBCO)
Action: moves current cursor position to an absolute position. Before calling, X co-ordinate must be in DE and the Y co-ordinate in HL. AF,BC,DE and HL register pairs are left corrupted.

GRA SET ORIGIN (CALL #BBC9)
Action: sets location of the graphics origin and moves cursor there. Before calling, the X co-ordinate must be in DE and the Y co-ordinate in HL.
AF, BC, DE and HL register pairs are left corrupted.

GRA WIN WIDTH (CALL #BBCF)
Action: sets right and left hand edges of the graphics window. Before calling, the X co-ordinate of one edge must be in DE and the other X co-ordinate in HL. (It doesn't matter which edge is which.)
AF, BC, DE and HL register pairs are left corrupted.

GRA WIN HEIGHT (CALL #BBD2)
Action: sets top and bottom edges of the graphics window.
Before calling, the Y co-ordinate of one window edge must be in DE and the Y co-ordinate of the other edge in HL. (It doesn't matter which edge is which.)
AF, BC, DE and HL register pairs are left corrupted.

GRA CLEAR WINDOW (CALL #BBDB)
Action: clear the graphics window to the current graphics paper ink.
AF, BC, DE and HL register pairs are left corrupted.

GRA SET PEN (CALL #BBDE)
Action: set the graphics plotting ink. Before calling, the ink number

must be in the accumulator.
A and F registers are left corrupted.

SET PAPER (CALL #BBE4)
Action: set the graphics background ink. Before calling, the ink number must be left in the accumulator.
A and F registers are left corrupted.

GRA PLOT ABSOLUTE (CALL #BBEA)
Action: plot a point at the absolute position relative to the user's origin. Before calling, the X co-rdinate must be in DE and the Y co-ordinate in HL.
AF, BC, DE and HL register pairs are left corrupted.

GRA PLOT RELATIVE (CALL #BBED)
Action: plot point relative to present co-ordinates. Before calling, the X offset co-ordinate must be in DE and the offset Y coordinate in HL. Either co-ordinates can be signed integers.
AF, BC, DE and HL register pairs are left corrupted.

GRA LINE ABSOLUTE (CALL #BBF6)
Action: draw line from present position to absolute co-ordinates supplied.
Before calling, the X co-ordinate must be in DE and the Y co-ordinate in HL.
AF, BC, DE and HL register pairs are left corrupted.

GRA LINE RELATIVE (CALL #BBF9)
Action: draw line from present position to relative co-ordinates supplied. Before calling, the X co-ordinate offset must be in DE and the Y co-ordinate offset must be in HL. Either co-ordinates can be signed integers.
AF, BC, DE and HL register pairs are left corrupted.

GRA WR CHAR (CALL #BBFC)
Action: put character on the screen at the current graphics cursor position. Before calling, the character must be in the accumulator.
AF, BC, DE and HL register pairs are left corrupted.

SCR SET BASE (CALL #BC08)
Action: set area of RAM used for screen memory. Before calling, the accumulator must contain the high order byte of the base address.
AF and HL register pairs are left corrupted.

SCR CLEAR (CALL #BC14)
Action: clears whole of screen memory to ink zero.
AF, BC, DE and HL register pairs are left corrupted.

SCR FILL BOX (CALL #BC44)
Action: fill a character area of the screen with an ink.

Before calling:
Ink number must be in accumulator.
H must contain the left column of the area.
D must contain the right column of the area.
L must contain the top row of the area.
E must contain the bottom row of the area.
AF, BC, DE and HL register pairs are left corrupted.
If illegal columns and/or rows are specified then weird results are possible.

8 Addition and subtraction

Addition and subtraction using 8,16 and 32 bit integers are described here. Multiplication and division will be held over to Chapter 10. During the course of this chapter, you may from time to time, find a need to refer back to the theory of signed and unsigned numbers discussed in Chapter 3.

Range limitations

For addition, we have the choice of ADD and ADC and for subtraction, SUB and SBC. Because of the inherent two's complement structure, they can add or subtract any mixture of positive and negative numbers. For example, ADD or ADC can add 4 to 7, −4 to 7, −4 to −7 etc. Similarly, SUB or SBC can subtract any mixture of signed numbers. Using two's complement, the largest positive number which any single length register can handle is 0111 1111 and the largest negative number is 1000 0000 (+127 and −128 respectively). Whether we intepret a result in two's complement or unsigned binary is purely a matter of choice. In unsigned integer form, the largest number is 255.

Although a memory location can only hold one byte, the Z80A does provide double byte addition and subtraction using register pairs. For example, **ADD HL,rp** will add the 16 bit number in a chosen register pair to the 16 bit number in HL. Note that HL effectively takes on the role of a 16 bit accumulator. Although any register pair can be chosen to hold one of the numbers, the other must be in HL and will always hold the result. A double length register can hold a signed integer range of −32,768 to +32,767 (65,535 in unsigned integer form).

Numbers higher than the limits imposed by double length registers can be handled but only by a series of instalments.

8 bit Addition and Subtraction

There are several ways of performing addition using the various

addressing methods available with the Z80A. In each example which follows, we assume that the two numbers to be added are present in locations #7100 and #7101. These locations are labelled 'first' and 'second' respectively. The result will be stored in address #7102. The examples can be tested if required by POKEing numbers into these locations from BASIC, CALLing the routine at address &7000 and PEEKing the result. For example:

```
POKE &7100,149
POKE &7101,27
CALL &7000
PRINT PEEK(&7102)
```

In practice, of course, the segment would not be used in this way but would be a small part of a more ambitious project.

Testing can alternatively be effected from the monitor, but remember to add an extra line:

```
ENT #7000
```

to the source code listing prior to assembly.

Example 8.1

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                       10 ;EXAMPLE 8.1
7100                   20 first:   EQU  #7100
7101                   30 second:  EQU  #7101
7102                   40 result:  EQU  #7102
7000                   50          ORG  #7000
7000   3A0171          60          LD   A,(second)
7003   47              70          LD   B,A
7004   3A0071          80          LD   A,(first)
7007   80              90          ADD  A,B
7008   320271         100          LD   (result),A
700B   C9             110          RET

Pass 2 errors: 00

Table used:     51   from    119
```

We start our examples by using direct addressing:

Lines 20 and 40: assign labels to locations used.
Line 50: forces assembly at address #7000.
Lines 60 and 70: load the B register with the second of the two

numbers to be added. Note that it is not possible to load the B register directly from memory so the accumulator is loaded first and the contents copied to the B register.
Line 80: loads the first number directly from memory into the accumulator.
Line 90: adds the two numbers together with the **ADD A,B** instruction. If subtraction is required this line can be changed to **SUB B.**
Line 100: stores the result, present in the accumulator, in memory using direct addressing.
Line 110: returns control back to the calling program. That is to say BASIC or the assembler.

Example 8.2

Although the direct addressing method works it is not the most efficient. The following example uses implied addressing and saves one byte of memory and consumes 4 less clock cycles.

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                      10 ;EXAMPLE 8.2
7100                  20 first:   EQU   #7100
7101                  30 second:  EQU   #7101
7102                  40 result:  EQU   #7102
7000                  50          ORG   #7000
7000  210171          60          LD    HL,second
7003  3A0071          70          LD    A,(first)
7006  86              80          ADD   A,(HL)
7007  320271          90          LD    (result),A
700A  C9             100          RET

Pass 2 errors: 00

Table used:    51  from   118
```

Line 60: with implied addressing, the HL register pair holds the address of the memory location accessed. The address of the second of the two numbers to be added is thus loaded into the HL pair.
Line 70: loads the first of the two numbers into the accumulator using direct addressing.
Line 80: adds the two numbers using implied addressing. If subtraction is required change this line to **SUB (HL).**

Example 8.3

A further improvement in efficiency can be effected by noting that the two 8 bit numbers to be added and the result are in sequential

locations. Example 8.3 saves a further 3 bytes of memory over Example 8.2.

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                        10 ;EXAMPLE 8.3
7100                    20 first:  EQU  #7100
7101                    30 second: EQU  #7101
7102                    40 result: EQU  #7102
7000                    50         ORG  #7000
7000  210071            60         LD   HL,first
7003  7E                70         LD   A,(HL)
7004  23                80         INC  HL
7005  86                90         ADD  A,(HL)
7006  23               100         INC  HL
7007  77               110         LD   (HL),A
7008  C9               120         RET

Pass 2 errors: 00

Table used:     51  from    120
```

Line 60: loads the address of the first of the two 8 bit numbers into the HL register pair which is used as a data pointer.
Line 70: loads the accumulator with the first number, using implied addressing.
Line 80: increments the data pointer, HL, to hold the address of the second number.
Line 90: adds the two numbers, using implied addressing and the result left in the accumulator. The necessary modification to perform subtraction is to replace the **ADD A,(HL)** instruction with **SUB (HL).**
Line 100: increments the data pointer HL, to hold the address of the result location.
Line 110: stores result in memory using implied addressing.

16 Bit Addition and Subtraction

In practice, 8 bit addition, although treated ad nauseum in text books (including this one), is sadly not used much. This is due to the severe limitations in number size that can be effectively handled (0 to 255). Even the designers of the Amstrad software decided on 16 bit integers as the standard for BASIC thus giving an improved integer arithmetic range of −32768 to +32767. Here we give a couple of examples of 16 bit addition and subtraction. In both examples the first and second numbers are assumed present in location pairs starting at #7100 and #7102 respectively. The result is stored in two sequential locations at address #7104. Remember the convention that the low order byte is always stored first. As before, the routines can be tested by POKEing values in from BASIC, CALLing the

routine at &7000 and PEEKing the result. However, since you can only POKE one byte at a time, care must be taken over setting up the numbers. Remember that the low byte contains units up to 256 and the high byte contains the number of 256s. The result can be printed out in BASIC by the following line:

PRINT PEEK(&7104)+PEEK(&7105)*256

Example 8.4

This example shows how to perform the addition or subtraction using implied addressing with simple 8 bit arithmetic instructions. Notice that the listing is much longer than the previous examples.

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                    10 ;EXAMPLE 8.4
7100                20 first:  EQU  #7100
7102                30 second: EQU  #7102
7104                40 result: EQU  #7104
7000                50         ORG  #7000
7000  110071        60         LD   DE,first
7003  210271        70         LD   HL,second
7006  1A            80         LD   A,(DE)
7007  86            90         ADD  A,(HL)
7008  320471       100         LD   (result),A
700B  23           110         INC  HL
700C  13           120         INC  DE
700D  1A           130         LD   A,(DE)
700E  8E           140         ADC  A,(HL)
700F  320571       150         LD   (result+1),A
7012  C9           160         RET

Pass 2 errors: 00

Table used:    51  from   127
```

Lines 60 and 70: load the DE and HL register pairs, acting as data pointers, with the addresses of the first and second numbers.
Lines 80 and 90: load the first number into the accumulator and adds it to the second, using implied addressing. Line 90 should be changed to **SUB (HL)** if subtraction is required.
Line 100: stores low byte result, present in the accumulator, in memory.
Lines 110 and 120: increment the data pointer DE and HL to point to the high bytes of 'first' and 'second'.
Lines 130 to 140: load the high byte of the first number into the accumulator and adds it to the high byte of the second number. Here we use the instruction **ADC A,(HL)**. This is because we need to add in the carry, if any, from the low byte addition. The

equivalent for subtraction would be the instruction **SBC A, (HL)** which could be substituted in line 140.

Line 150: stores the high byte of the result in memory. Notice the use of result+1 here. The label 'result' is assigned a value #7104. By informing the assembler that we wish to store the accumulator contents at address result+1, the value #7105 is evaluated.

With the Z80 we are fortunate, unlike that of 6502 based micros, that arithmetic can be performed on register pairs. This luxury, has the advantage of freeing us from the worry of carries from low to high bytes when working with 16 bit numbers. The following instructions can be used directly on 16 bit numbers:

ADD HL,rp
ADC HL,rp
SBC HL,rp

If you are wondering why SUB HL,rp is not included, don't worry, there isn't one. However, it can be mimicked by the following two lines:

AND A
SBC HL,rp

The AND A instruction, or in fact any boolean instruction, simply clears the carry flag.

Example 8.5 shows a shorter and faster method of 16 bit addition or subtraction.

Example 8.5

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                     10 ;EXAMPLE 8.5
7100                 20 first:   EQU  #7100
7102                 30 second:  EQU  #7102
7104                 40 result:  EQU  #7104
7000                 50          ORG  #7000
7000  2A0071         60          LD   HL,(first)
7003  ED5B0271       70          LD   DE,(second)
7007  19             80          ADD  HL,DE
7008  220471         90          LD   (result),HL
700B  C9            100          RET

Pass 2 errors: 00

Table used:     51   from   119
```

Lines 60 and 70: load the HL and DE register pairs directly from memory by means of the 16 bit load instructions. The low bytes of the two numbers to be added are loaded into the L and E registers. The high bytes are loaded into the H and D registers.

Line 80: performs the addition. Replace with **AND A** followed by **SBC HL,DE** for subtraction.
Line 90: the 16 bit load instruction stores the result, present in the register pair HL, into the location #7104 (low byte) and #7105 (high byte).

As a general rule it is more efficient to use 16 bit register pair loads where possible. However, there are always exceptions as you will probably find out when tackling more ambitious programs.

32 Bit Addition and Subtraction

There are a few machines, for example the BBC micro, that perform 32 bit integer arithmetic in BASIC. If we wish to handle such large numbers we will have to set aside 4 bytes for each number. Here we will give two examples each with its own merit. In both cases the first number will be assumed to occupy four bytes starting at #7100, the second number at #7104 and the result will be stored in four bytes starting at #7108. Although it is possible to test out the routines as they stand by PEEKing and POKEing from BASIC it is obviously going to be a little on the tedious side. It will probably be more rewarding to simply study the coding and accept that it works OK in practice.

Example 8.6

Example 8.6, although economical on memory, is fairly heavy on the use of clock cycles (execution time). This stems from the fact that a loop is executed four times to add the bytes. With assembly language programming you are often faced with this sort of dilemma. Should I use the fastest coding or the more economical on memory? is a question often posed. The answer of course depends on many factors. For example, in a short routine or animation sequence execution speed is of prime importance. On the other hand, if memory is tight, speed may be of secondary importance in certain areas of a program. In Example 8.6 one of the original numbers to be added is overwritten by the result but this will not be a problem in the majority of cases.

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                      10 ;EXAMPLE 8.6
7100                  20 first:  EQU  #7100
7104                  30 second: EQU  #7104
7108                  40 result: EQU  #7108
7000                  50         ORG  #7000
7000  110071          60         LD   DE,first
7003  210471          70         LD   HL,second
7006  A7              80         AND  A
```

```
7007  0604        90         LD   B,4
7009  1A         100 loop:   LD   A,(DE)
700A  8E         110         ADC  A,(HL)
700B  77         120         LD   (HL),A
700C  23         130         INC  HL
700D  13         140         INC  DE
700E  10F9       150         DJNZ loop
7010  C9         160         RET

Pass 2 errors: 00

Table used:     62  from    126
```

Lines 60 and 70: load the register pairs DE and HL respectively with address of the first and second numbers to be added.
Line 80: clears the carry flag.
Line 90: initialises the loop counter, register B to 4.
Lines 100 to 150: add the four bytes of each 32 bit number using implied addressing. Notice that ADC is used each time to add in the carry between bytes. Since we cleared the carry flag in line 80 before entering the loop we do not need to use the ADD instruction on the first byte as in previous examples. Again, replace line 110 with **SBC A,(HL)** if subtraction is needed.

Example 8.7

Example 8.7, although more economical on the use of clock cycles, is heavier on memory requirements. Here, we use 16 bit register pair loads and arithmetic using direct addressing. In view of this we only need to worry about carries from the 2nd to 3rd bytes. The other carries are taken into account automatically by the 16 bit arithmetic instructions.

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                   10 ;EXAMPLE 8.7
7100               20 first:  EQU  #7100
7104               30 second: EQU  #7104
7108               40 result: EQU  #7108
7000               50         ORG  #7000
7000  2A0071       60         LD   HL,(first)
7003  ED5B0471     70         LD   DE,(second)
7007  19           80         ADD  HL,DE
7008  220871       90         LD   (result),HL
700B  2A0271      100         LD   HL,(first+2)
700E  ED5B0671    110         LD   DE,(second+2)
7012  ED5A        120         ADC  HL,DE
7014  220A71      130         LD   (result+2),HL
7017  C9          140         RET

Pass 2 errors: 00

Table used:     51  from    127
```

Lines 60 to 70: load the register pairs HL and DE with the lower two bytes of the numbers to be added using direct addressing.
Line 80: adds the two 16 bit values together and updates the carry flag according to the result. (Use **AND A** followed by **SBC HL,DE** for subtraction). It is worth stressing again that there is no SUB HL,DE instruction in the Z80 repertoire.
Line 90: stores the 16 bit result of the addition in memory.
Lines 100 and 110: since the lower two bytes of the addition have now been calculated we need to load the HL and DE register pairs with the 3rd and 4th bytes of the two numbers to be added. This is why two is added to the labels first and second. For example, the label 'first' is assigned an address #7100. By specifying first+2 as a label the assembler evaluates it to #7102.
Line 120: adds the 3rd and 4th bytes of both numbers, now in HL and DE. Notice that an ADC instruction is used here so the carry, if any, from the 1st and 2nd byte result is added in. Use **SBC HL,DE** here if subtraction is required rather than addition.
Line 140: stores the resultant 3rd and 4th bytes in memory at address #710A.

Summary

1 Only the accumulator can hold the result of single byte arithmetic and only HL can hold the result of double byte arithmetic.
2 ADD is normally used for single byte addition although ADC can be used providing the carry is cleared first.
4 In double byte addition, ADD is normally used for the first byte and ADC for the second byte.
5 There is no SUB HL,rp instruction.
6 AND A can be used at any time to clear the carry flag without corrupting the contents of the accumulator.
7 Numerically, each bit in a high byte is worth 256 times as much as the corresponding bit in the low byte.

Decision making and loop structures

9

In BASIC, and many other high level languages, we have a limited range of decision making commands and loop structures. With machine code we have considerably more freedom. This chapter treats the more commonly used structures. After a preliminary read through, it should be used as a reference guide.

Branch on decision structures

Decision making in machine code involves the interrogation of status flags, particularly the carry flag (C) and the zero flag (Z). These are updated by most arithmetic or logical instructions and are subsequently used as the condition within JP or JR type instructions.

Testing for zero

After performing some arithmetic or logical process it will often be necessary to test some register, usually the accumulator, for zero. If a segment of code or 'process' is to be skipped by, say, a branch to a location labelled 'over', the flag status is tested as follows:

(a) To test for zero, use **JR Z,over**
(b) To test for non zero, use **JR NZ,over**

There are occasions when the flags need to be set according to the contents of the accumulator. We should remember that load instructions have no effect on the status flags as is common with such microprocessors as the 6502. In such cases the following sequence may be used to update the status flags.

```
       LD A,(number)
       AND A
       JR Z,over
       - - - - - - -
       - - - - - - -
       - - - - - - -
over:  - - - - - - -
```

Note that the AND instruction sets the status flags without affecting the contents of the accumulator.

Greater care is needed where the condition involves the relationship between two numbers because it will involve subtracting, or comparing, one number with the other. This means that the result of the subtraction, and hence the flag status, will depend on *which way round* it is performed. That is to say, whether the first number is subtracted from the second or vice versa. With SUB,SBC and CP instructions the subtraction is always FROM the accumulator. In the case of register pair subtraction, the subtraction is FROM the HL register.

This leads to the following guide:

1 Use **JR C** to branch on 'less-than' decisions. (Still use **JR C** on 'greater-than decisions' but subtract, or compare, the numbers the opposite way round.)
2 Use **JR NC** to branch on 'less than or equal to' decisions. (Still use **JR NC** on 'greater than or equal to' decisions but subtract, or compare, the two numbers the opposite way round.

It is worth reminding you that comparison instructions, such as CP, leave the contents of registers or memory locations undisturbed. The only effect of CP is on the flags so it is always intended to be followed by a conditional branch or jump.

The following examples deserve careful study because they provide examples covering the most used cases. It is assumed that the labelled locations, 'first', 'second' and 'number', have been assigned earlier. Naturally, they are not intended to be run in their present skeletal state but should be used as reference material. For simplicity, direct addressing has been used in the examples. However, the same general principles will apply in all the other addressing modes.

You will note that the line, **AND A,** often appears. One use has already been discussed but another is simply a method of clearing the carry flag before a **SBC HL,rp** instruction. There is no direct instruction in the Z80A for clearing the carry flag so the **AND A** instruction or any other logical instruction is used instead. Note that there is no **SUB HL,rp** instruction in the Z80 repertoire.

A problem arises when decision making after a register pair increment or decrement instruction because the flags are not updated on the result. This deficiency was a relic of the earlier 8080 microprocessor and, in the interests of software compatibility, was not changed in its Z80 offspring. Nevertheless, we can easily update the flags by using one of the logical instructions. An often used method of decrementing, and checking whether the contents of a register pair is zero, in this case BC, is as follows:

```
DEC BC
LD A,B
OR C
JR NZ,label
```

How it works may not be immediately obvious. We first load B, the high byte of the register pair BC, into the accumulator. We then logically OR the lower byte of the register pair (the C register) with the accumulator. The accumulator will only yield a zero result if neither of the participants contained any binary '1's. At the same time the OR instruction, like all other Boolean instructions, updates the status flags according to the result. A branch can then be made according to the result of the Z flag.

Example 9.1
Objective: branch to 'over' if 'number' in accumulator = zero.

```
        LD A, (number)
        AND A
        JR Z,over
        - - - - - - - -
        - - - - - - - -
over    - - - - - - - -
```

Example 9.2
Objective: Branch to 'over' if 'number' in accumulator is non zero.

```
        LD A, (number)
        AND A
        JR NZ,over
        - - - - - - - -
        - - - - - - - -
over:   - - - - - - - -
```

Example 9.3
Objective: Branch to 'over' if 'number' in register pair = zero.

```
        LD BC, (number)
        LD A,B
        OR C
        JR Z,over
        - - - - - - - -
        - - - - - - - -
        - - - - - - - -
over:   - - - - - - - -
```

Example 9.4
Objective: Branch to 'over' if 'number' in register pair = non zero.

```
        LD BC, (number)
        LD A,B,
        OR C
        JR NZ,over
        - - - - - - - -
        - - - - - - - -
over:   - - - - - - - -
```

Example 9.5

Objective: branch to label 'over' if 'first' single byte number = 'second' single byte number.

```
        LD A,(second)
        LD B,A
        LD A, (first)
        CP B
        JR Z,over
        – – – – – – – –
        – – – – – – – –
over:   – – – – – – – –
```

Example 9.6

Objective: branch to label 'over' if 'first' double byte number = 'second' double byte number.

```
        LD HL, (first)
        LD DE, (second)
        AND A
        SBC HL,DE
        JR Z,over
        – – – – – – – –
        – – – – – – – –
over:   – – – – – – – –
```

Example 9.7

Objective: branch to 'over' if 'first' number is not equal to 'second' number.

```
        LD A,(second)
        LD B,A
        LD A,(first)
        CP B
        JR NZ,over
        – – – – – – – –
        – – – – – – – –
over:   – – – – – – – –
```

Example 9.8

Objective: branch to 'over' if 'first' double byte number is not equal to, 'second' double byte number.

```
        LD HL,(first)
        LD DE,(second)
```

```
        AND A
        SBC HL,DE
        JR NZ,over
        - - - - - - - -
        - - - - - - - -
over:   - - - - - - - -
```

Example 9.9
Objective: branch to 'over' if 'first' number is greater than or equal to 'second' number.

```
        LD A,(second)
        LD B,A
        LD A,(first)
        CP B
        JR NC,over
        - - - - - - - -
        - - - - - - - -
over:   - - - - - - - -
```

Example 9.10
Objective: branch to 'over' if 'first' double byte number is greater than, or equal to, 'second' double byte number.

```
        LD HL,(first)
        LD DE,(second)
        AND A
        SBC HL,DE
        JR NC,over
        - - - - - - - -
        - - - - - - - -
over:   - - - - - - - -
```

Example 9.11
Objective: branch to 'over' if 'first' number is less than 'second' number.

```
        LD A,(second)
        LD B,A
        LD A,(first)
        CP B
        JR C,over
        - - - - - - - -
        - - - - - - - -
over:   - - - - - - - -
```

Example 9.12
Objective: branch to 'over' if 'first' double byte number is less than 'second' double byte number.

```
       LD HL,(first)
       LD DE,(second)
       AND A
       SBC HL,DE
       JR NC,over
       - - - - - - - -
       - - - - - - - -
over:  - - - - - - - -
```

Example 9.13
Objective: branch to 'over' if 'first' number is greater than 'second' number.

```
       LD A,(first)
       LD B,A
       LD A,(second)
       CP B
       JR C,over
       - - - - - - - -
       - - - - - - - -
over:  - - - - - - - -
```

Example 9.14
Objective: branch to 'over' if 'first' double byte number is less than 'second' double byte number.

```
       LD DE, (first)
       LD HL,(second)
       AND A
       SBC HL,DE
       JR C,over
       - - - - - - - -
       - - - - - - - -
over:  - - - - - - - -
```

Example 9.15
Objective: branch to 'over' if 'first' number is less than, or equal to, 'second' number.

```
       LD A,(first)
       LD B,A
       LD A,(second)
       CP B
```

```
        JR NC,over
        - - - - - - - -
        - - - - - - - -
over:   - - - - - - - -
```

Example 9.16

Objective: branch to 'over' if 'first' double byte number is less than, or equal to, 'second' double byte number.

```
        LD HL,(second)
        LD DE,(first)
        AND A
        SBC HL,DE
        JR NC,over
        - - - - - - - -
        - - - - - - - -
over:   - - - - - - - -
```

Simple loop structures

Loops are a commonly used structure and can be either upcounting or downcounting. Of the two, downcounting loops are the more efficient since fewer instructions are needed. This will become evident on studying the later examples.

A loop performs a repetitive process and consists of:

(a) The loop control counter for defining the number of times the loop revolves. For down-count loops, the control counter starts high and is decremented each time round towards zero. For up-count loops, it starts low, usually zero, and is incremented each time round.
(b) The actual process.
(c) The end of loop test. This can either be made at the start of the process (test limit first type) or more commonly, after the process (test limit last type). For downcounting loops the loop counter is normally tested for zero each cycle. With upcounting loops the loop counter is compared with some end value.

For simplicity, the 'process' within the following loops is one which displays the '*' character on the screen. In practice, of course, the process can be complex and extend over many lines. The following examples 9.17 to 9.25, although of no immediate practical use, can be entered, assembled and executed directly. In this way, familiarity with loop structures may be gained without detracting from the essential details. Since the ENT assembler directive is used, the object code can be executed simply by entering 'R' from within the assembler. However, the code can also be called from BASIC with CALL &7000. Where possible both 8 bit and 16 bit count loops are given.

Example 9.17

Objective: repeat a process until the ENTER key is pressed.

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                          10 ;EXAMPLE 9.17
BB5A                      20 OUTPUT: EQU  #BB5A
BB09                      30 RCHAR:  EQU  #BB09
7000                      40         ENT  #7000
7000                      50         ORG  #7000
7000   CD09BB             60 loop:   CALL RCHAR
7003   FE0D               70         CP   #0D
7005   2807               80         JR   Z,exit
7007   3E2A               90         LD   A,"*"
7009   CD5ABB            100         CALL OUTPUT
700C   18F2              110         JR   loop
700E   C9                120 exit:   RET

Pass 2 errors: 00

Table used:    60  from   120
Executes: 28672
```

A flowchart is given in Figure 9.1. Note the call to RCHAR is an operating system subroutine for placing the ASCII character from the keyboard into the accumulator and is previously assigned to address #BB09.

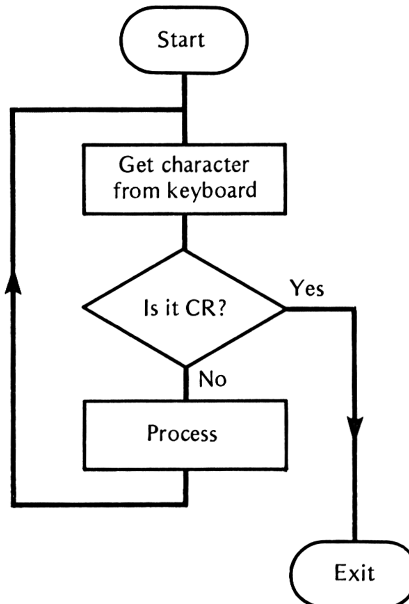


Fig 9.1 Flow chart for Example 9.17

Example 9.18
Objective: Single byte upcounting loop.

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                      10 ;EXAMPLE 9.18
BB5A                  20 OUTPUT: EQU  #BB5A
7000                  30         ENT  #7000
7000                  40         ORG  #7000
7000   0E10           50         LD   C,#10
7002   0600           60         LD   B,0
7004   3E2A           70 loop:   LD   A,"*"
7006   CD5ABB         80         CALL OUTPUT
7009   04             90         INC  B
700A   79            100         LD   A,C
700B   B8            110         CP   B
700C   20F6          120         JR   NZ,loop
700E   C9            130         RET

Pass 2 errors: 00

Table used:     37  from    119
Executes: 28672
```

Lines 50 and 60 set the loop parameters. They are given arbitrary numbers but can be altered as desired within the upper limit of $FF (255 decimal). The process will always be executed at least once, whatever the loop parameters. See Figure 9.2 for flowchart.

Example 9.19
Objective: double byte upcounting loop.

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                      10 ;EXAMPLE 9.19
BB5A                  20 OUTPUT: EQU  #BB5A
7000                  30         ENT  #7000
7000                  40         ORG  #7000
7000   010000         50         LD   BC,0
7003   110004         60         LD   DE,#400
7006   3E2A           70 loop:   LD   A,"*"
7008   CD5ABB         80         CALL OUTPUT
700B   03             90         INC  BC
700C   7B            100         LD   A,E
700D   B9            110         CP   C
700E   20F6          120         JR   NZ,loop
7010   7A            130         LD   A,D
7011   B8            140         CP   B
7012   20F2          150         JR   NZ,loop
7014   C9            160         RET
```

```
Pass 2 errors: 00

Table used:     37  from   123
Executes: 28672
```

Lines 50 and 60 set the loop parameters. They are given arbitrary numbers but can be altered as desired within the upper limit of #FFFF (65,535 decimal). The process will always be executed at least once, whatever the loop parameters. See Figure 9.2 for flowchart.

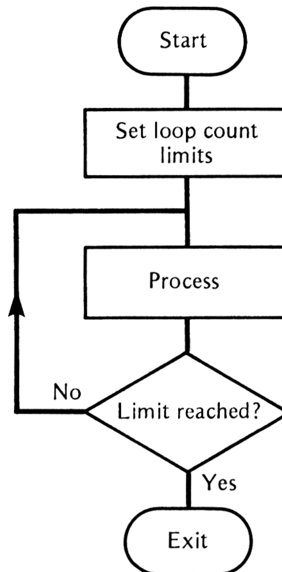


Fig 9.2 Flowchart for 'test limit last' type loops

Example 9.20
Objective: Single byte downcounting loop.

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                        10 ;EXAMPLE 9.20
BB5A                    20 OUTPUT: EQU  #BB5A
7000                    30         ENT  #7000
7000                    40         ORG  #7000
7000   06FF             50         LD   B,#FF
7002   3E2A             60 loop:   LD   A,"*"
7004   CD5ABB           70         CALL OUTPUT
7007   10F9             80         DJNZ loop
7009   C9               90         RET

Pass 2 errors: 00

Table used:     37  from   115
Executes: 28672
```

The first two lines sets the loop counter. They are given arbitrary numbers but can be altered as desired within the upper limit of #FF (255 decimal). The process will always be executed at least once, whatever the loop parameters. See Figure 9.2 for flowchart.

Example 9.21
Objective: double byte downcounting loop.

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                      10 ;EXAMPLE 9.21
BB5A                  20 OUTPUT: EQU  #BB5A
7000                  30         ENT  #7000
7000                  40         ORG  #7000
7000  010004          50         LD   BC,#400
7003  3E2A            60 loop:   LD   A,"*"
7005  CD5ABB          70         CALL OUTPUT
7008  0B              80         DEC  BC
7009  78              90         LD   A,B
700A  B1             100         OR   C
700B  20F6           110         JR   NZ,loop
700D  C9             120         RET

Pass 2 errors: 00

Table used:    37  from   118
Executes: 28672
```

Line 50 sets the loop parameter. The number used is quite arbitrary but can be altered as desired within the upper limit of #FFFF (65,535 decimal). The process will always be executed at least once, whatever the loop parameters. See Figure 9.2 for flowchart.

Example 9.22
Objective: Single byte upcounting loop.

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                      10 ;EXAMPLE 9.22
BB5A                  20 OUTPUT: EQU  #BB5A
7000                  30         ENT  #7000
7000                  40         ORG  #7000
7000  0EFF            50         LD   C,#FF
7002  0600            60         LD   B,0
7004  78              70 loop:   LD   A,B
7005  B9              80         CP   C
7006  2808            90         JR   Z,over
7008  3E2A           100         LD   A,"*"
```

```
700A  CD5ABB     110         CALL OUTPUT
700D  04         120         INC  B
700E  18F4       130         JR   loop
7010  C9         140 over:   RET

Pass 2 errors: 00

Table used:    48  from   121
Executes: 28672
```

Lines 50 and 60 set the loop parameters. They are given arbitrary numbers but can be altered as desired within the upper limit of #FF (255 decimal). Unlike the previous examples the process within the loop will not necessarily be executed at all. (For example, if the two loop limits are the same). See Figure 9.3 for flowchart.

Example 9.23
Objective: double byte-upcounting loop.

```
Hisoft GENA3.1 Assembler. Page    1.

Pass 1 errors: 00

                  10 ;EXAMPLE 9.23
BB5A              20 OUTPUT: EQU  #BB5A
7000              30         ENT  #7000
7000              40         ORG  #7000
7000  110004      50         LD   DE,#400
7003  010000      60         LD   BC,0
7006  7B          70 loop:   LD   A,E
7007  B9          80         CP   C
7008  2004        90         JR   NZ,proc
700A  7A         100         LD   A,D
700B  B8         110         CP   B
700C  2808       120         JR   Z,over
700E  3E2A       130 proc:   LD   A,"*"
7010  CD5ABB     140         CALL OUTPUT
7013  03         150         INC  BC
7014  18F0       160         JR   loop
7016  C9         170 over:   RET

Pass 2 errors: 00

Table used:    59  from   125
Executes: 28672
```

Lines 50 and 60 set the loop parameters. They are given arbitrary numbers but can be altered as desired within the upper limit of #FFFF (65,535 decimal). See Figure 9.3 for flowchart.

Example 9.24
Objective: Single byte down-counting loop.

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                       10 ;EXAMPLE 9.24
BB5A                   20 OUTPUT: EQU  #BB5A
7000                   30         ENT  #7000
7000                   40         ORG  #7000
7000  06FF             50         LD   B,#FF
7002  78               60 loop:   LD   A,B
7003  A7               70         AND  A
7004  2808             80         JR   Z,over
7006  3E2A             90         LD   A,"*"
7008  CD5ABB          100         CALL OUTPUT
700B  05              110         DEC  B
700C  18F4            120         JR   loop
700E  C9              130 over:   RET

Pass 2 errors: 00

Table used:    48  from    120
Executes: 28672
```

Line 50 sets the loop parameters. They are given an arbitrary number but can be altered as desired within the upper limit of #FF (255 decimal). See Figure 9.3 for flowchart.

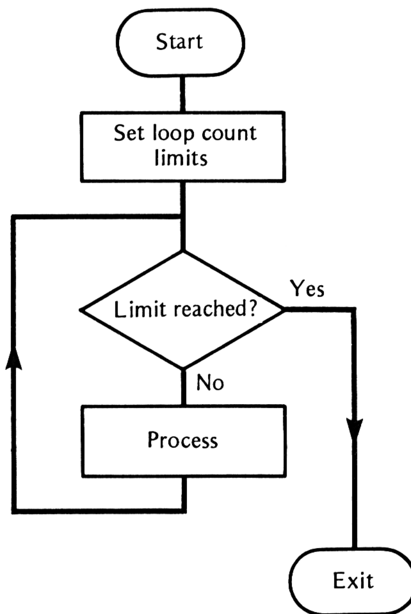


Fig 9.3 Flow chart for 'test limit first' type loop

Example 9.25
Objective: double byte downcounting loop.

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                        10 ;EXAMPLE 9.25
BB5A                    20 OUTPUT: EQU   #BB5A
7000                    30         ENT   #7000
7000                    40         ORG   #7000
7000  010004            50         LD    BC,#400
7003  78                60 loop:   LD    A,B
7004  B1                70         OR    C
7005  2808              80         JR    Z,over
7007  3E2A              90         LD    A,"*"
7009  CD5ABB           100         CALL  OUTPUT
700C  0B               110         DEC   BC
700D  18F4             120         JR    loop
700F  C9               130 over:   RET

Pass 2 errors: 00

Table used:    48  from   120
Executes: 28672
```

Line 50 sets the loop parameter. They are given an arbitrary number but can be altered as desired within the upper limit of #FFFF (65,535 decimal). See Figure 9.3 for flowchart.

Preserving register contents

Although the Z80 is well equipped with registers, situations often arise where you would like a few more. For example, setting up a loop, particularly a double byte variety, will often require one or more registers for use within the process. This means that the control counter or other loop parameters must be stored temporarily before entering the process. After the process, the register contents must be replaced again so that the loop can continue. The following example assumes that BC and DE are worth saving.

Example 9.26
Objective: freeing registers for use within the 'process'.

PUSH BC
PUSH DE

"
"
process
"
"

```
POP DE
POP BC
```

The stack can be used for temporary storage but remember that it acts as a LIFO memory so, as you can see from above, you must POP the data back in reverse order.

Branching according to sign

Before proceeding further, we suggest that you refer back to the paragraph 'Unsigned binary or two's complement' in Chapter 3 and, in particular, Figure 3.3. Bit 7 in a register or memory location is the two's complement sign bit. It is 1 for negative and 0 for positive numbers. In double, or multiple byte numbers, only bit 7 in the higher order byte represents the sign, so bit 7 in the lower order byte has no sign significance.

Examination of the Z80A instruction set reveals certain difficulties associated with conditional jump instructions. For example, it is not possible to use a JR conditional jump when the condition depends on the status of the sign bit. This is a pity because the advantage of ease of relocation is lost. That is to say, the ability of a program to still function correctly when transferred to another area of memory. The subject of relocation will be discussed more fully in a later chapter but, in the meantime, note that JR instructions, because they use relative adddressing, present no relocation problems.

Using JP P and JP M instructions

Returning to the problem of testing the sign bit, it can be seen from the instruction set that one solution is to make use of either **JP M,label** (jump on Minus) or **JP P,label** (jump on Plus).

Unfortunately, JP instructions use absolute addressing which means they are not directly relocatable and require a 2 byte jump address. If such disadvantages can be tolerated, they provide an easy solution to the problem.

Using left shift for testing the sign

LD instructions have no effect on the flags so when a memory location is loaded into the accumulator some artifact must be used in order to establish the status of bit 7. One way is to shift the accumulator left so that bit 7 enters the Carry flag. If the carry is set to 1, we can use **JR C** to branch if negative or **JR NC** to branch on positive. This method has the advantage of relative addressing but the shift action corrupts the accumulator although this may not always be a disadvantage.

Using the BIT test

The instruction **BIT n,A** puts the complement of bit n in the Z flag but has no effect on the accumulator contents. After **BIT 7,A** has been executed, the Z flag will be '1' if bit 7 was a '0'. If this is followed by **JR Z,label** the jump occurs only if the number was positive. This allows the use of a **JR Z** or **JR NZ** instruction for branching according to sign. Although the test for sign is the more common application, **BIT n,A** can be used to test the status of any bit. For example, **BIT 3,A** will set the Z flag according to the status of bit 3.

Example 9.27
Objective: absolute branch to 'pos' if location 'mem' holds a positive number.

```
LD A,(mem)
AND A
JP P,pos
```

Disadvantage: not directly relocatable.

Example 9.28
Objective: absolute branch to 'neg' if location 'mem' holds a negative number.

```
LD A,(mem)
AND A
JP M,neg
```

Disadvantage: not directly relocatable.

Example 9.29
Objective: relative branch to 'pos' if location 'mem' holds a positive number.

```
LD A,(mem)
RL A
JR NC,pos
```

Disadvantage: corrupts accumulator contents.

Example 9.30
Objective: relative branch to 'neg' if location 'mem' holds a negative number.

```
LD A,(mem)
RL A
JR C,neg
```

Disadvantage: corrupts accumulator contents.

Example 9.31
Objective: relative branch to 'pos' if location 'mem' holds a positive number.

```
LD A,(mem)
BIT 7,A
JR Z,pos
```

This is the preferred method.

Example 9.32
Objective: relative branch to 'pos' if location 'mem' holds a positive number.

```
LD A,(mem)
BIT 7,A
JR NZ,pos
```

This is the preferred method.

Summary

1 CP instructions have no effect other than updating the C,Z,S and Ac flags.
2 LD, DEC rp or INC rp instructions have no effect on flags.
3 Using AND A as a dummy instruction is useful for updating the status flags.
4 If the end-of-loop test is made after the process, the loop executes at least once. If made before, the loop may not necessarily execute at all.
5 Whe using the stack as a temporary storage medium, POPs must be in reverse order to PUSHes.
6 JR instructions are normally preferable to JP instructions because they are directly relocatable.
7 Direct jumps, conditional to the status of the sign bit, can only be made with JP instructions.
8 Left shift can be used after an LD instruction to test the sign bit but the accumulator contents are corrupted.
9 The BIT test is used to set the Z flag according to the status of any bit in a register or memory.

10 Multiplication and division

Flow charts are provided to aid understanding of the algorithms used in this treatment of 8 and 16 bit unsigned and 16 bit signed multiplication and division.

Multiplication

Performing decimal multiplication with pencil and paper is essentially an artificial process, relying on your memory of multiplication tables. In the case of older readers, this will very well depend on how often you were thumped at school during the morning table chanting. Without a calculator handy, the answer to 'What is 9×7?' can only be answered instantly if we remember our tables. If not, we have to hide our shame and laboriously keep adding nines together. You may be relieved to know that multiplication in binary is relatively easy because of the stark simplicity of the multiplication table. There are no tables to learn since no single binary digit can exceed 1. This is all we have to remember:

$1 \times 0 = 0$ and $1 \times 1 = 1$

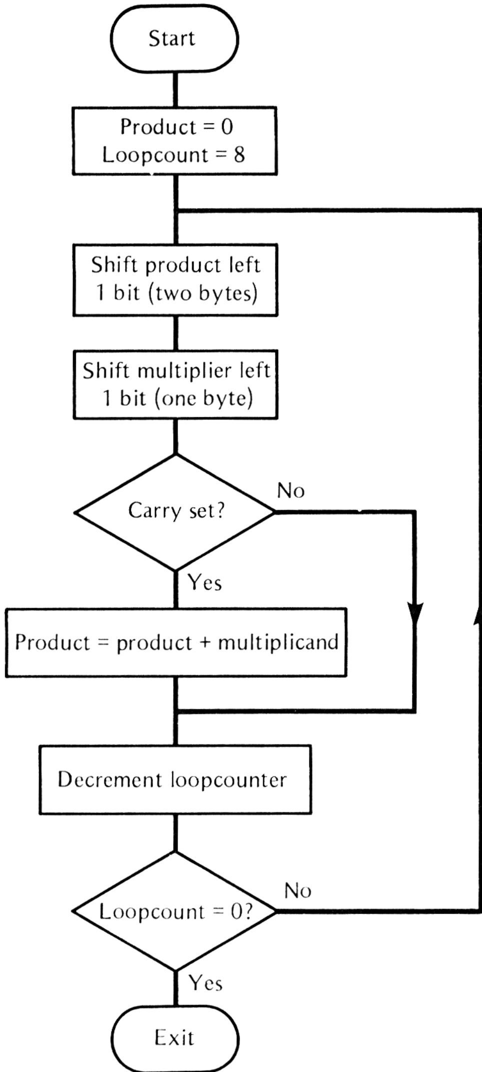
From this, we can deduce that:

(a) When a multiplier bit is 0, no action is needed.
(b) Action is only needed when a multiplier bit is a 1.
(c) Multiplication by 1 doesn't alter a number so any intermediate results need only be added to a shifted partial product each time.

Before going into detail we need a few definitions. When two numbers are to be multiplied together, one of them is officially known as the *multipicand*, the other number is the *multiplier* and the result of the multiplication is known as the *product*. It doesn't matter which of the two numbers you call the multiplicand and which you call the multiplier. One annoying trait of multiplication is the size of the product. Allowing for the worst case in single byte numbers, 255 mutliplied by 255 gives a product of 65,025. Such a number can only be accommodated if two bytes are considered placed end to end.

The program which follows is capable of multiplying any pair of 8-bit numbers and leaving the 16 bit product in two adjacent memory locations.

Unsigned 8 bit multiplication



The algorithm is similar in many respects to the pencil and paper method of binary multiplication taught in many schools. After preliminary initialisation of the product to zero, an eight-cycle loop is entered. (The loopcounter is set to the number of bits in the multiplier.) The loop begins with shifting the two byte product left. This staggers the columns in a similar way to that used in the simple pencil and paper method. Next, the single byte multiplier is shifted left. If the msb of the multiplier was a 1, the carry would have been set and the multiplicand is added to the partial product. If the msb was a 0, the carry would be a 0 so no addition is required. (Remember from the binary multiplication table above that multiplication by 0 is a useles exercise). The procedure is repeated for a total of 8 cycles, during which the 16 bit product builds up.

Simple 8-bit multiplication

Program 13.1 shows one method of coding the flow chart, Figure 10.1.

Example 10.1 Simple 8-bit multiplication

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                   10 ;UNSIGNED 8 BIT MULTIPLICATION
7000               20 begin:  EQU  #7000
7100               30 top:    EQU  begin+#100
7100               40 number: EQU  top
7101               50 mult:   EQU  top+1
7102               60 prod:   EQU  top+2
7000               70         ORG  begin
7000  3A0071       80         LD   A,(number)
7003  5F           90         LD   E,A
7004  3A0171      100         LD   A,(mult)
7007  210000      110         LD   HL,0
700A  55          120         LD   D,L
700B  0608        130         LD   B,8
700D  29          140 shift:  ADD  HL,HL
700E  17          150         RLA
700F  3001        160         JR   NC,over
7011  19          170         ADD  HL,DE
7012  10F9        180 over:   DJNZ shift
7014  220271      190         LD   (prod),HL
7017  C9          200         RET

Pass 2 errors: 00

Table used:    93  from   136
```

How example 10.1 works

This example assumes that the multiplicand and multiplier are present in locations #7100 and #7101 respectively. The product will be stored in two bytes starting #7102. Thereby the program can be used as a subroutine and called by another machine code program. It is assembled at the starting address #7000 so it can be tested from BASIC by POKEing in values, executing with **CALL &7000** and PEEKing the resulting product.

There now follows a line by line analysis.

Lines 20 to 60: assign labels to locations. The multiplicand is labelled 'number' and the multiplier 'mult'.

Line 70: is the pseudo instruction to the assembler, informing it where to assemble the object code.

Lines 80 and 90: load the E register with the 8 bit multiplicand (number).

Line 100: loads the accumulator with the 8 bit multiplier (mult).

Lines 110 and 120: initialise the register pair HL (used to temporarily hold the partial product) and the D register to zero. The D register corresponds to the unused high byte of the 8 bit multiplicand and is cleared so that 16 bit register pair arithmetic can be used in line 170.

Line 130: initialises the loopcounter to the number of bits in the multiplier.

Line 140: this instruction acts as a 16 bit shift left intruction on the partial product (HL register pair).

Line 150: rotate the multiplier left one bit so that the most significant bit is present in the carry status flag. Since it does not matter about garbage entering in from the lsb end of the accumulator the **RLA** instruction is used in preference to the more obvious **SLA A** instruction.

Line 160: performs the crucial test on the carry bit. The carry is clear only if the bit shifted out from the multiplier is a zero, in which case no addition is required and the instruction in line 170 is skipped.

Line 180: While the loop counter remains non zero, the top of the loop labelled 'shift' is repeatedly re-entered. When it reaches zero, all bits of the multiplier have been examined and the loop can be vacated.

Line 170: adds the multiplicand present in DE to the current product in the HL register pair.

Unsigned 16 bit multiplication

Example 10.2 allows for the fact that two 16 bit numbers multiplied together could, under worst case conditions, require four bytes to hold the product.

Example 10.2 Multiplication of 16 bit unsigned numbers

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00
                      10 ;UNSIGNED 16 BIT MULTIPLICATION
7000                  20 begin:   EQU  #7000
7100                  30 top:     EQU  begin+#100
7100                  40 number:  EQU  top
7102                  50 mult:    EQU  top+2
7104                  60 prod:    EQU  top+4
7000                  70          ORG  begin
7000  97              80          SUB  A
7001  4F              90          LD   C,A
7002  5F             100          LD   E,A
7003  57             110          LD   D,A
7004  0610           120          LD   B,16
7006  2A0271         130          LD   HL,(mult)
7009  CB23           140 shift:   SLA  E
700B  CB12           150          RL   D
700D  CB11           160          RL   C
700F  17             170          RLA
7010  29             180          ADD  HL,HL
7011  300D           190          JR   NC,over
7013  E5             200          PUSH HL
7014  2A0071         210          LD   HL,(number)
7017  19             220          ADD  HL,DE
7018  EB             230          EX   DE,HL
7019  E1             240          POP  HL
701A  3004           250          JR   NC,over
701C  0C             260          INC  C
701D  2001           270          JR   NZ,over
701F  3C             280          INC  A
7020  10E7           290 over:    DJNZ shift
7022  47             300          LD   B,A
7023  ED530471       310          LD   (prod),DE
7027  ED430671       320          LD   (prod+2),BC
702B  C9             330          RET

Pass 2 errors: 00

Table used:     93  from    152
```

To use the routine, the two byte multiplicand must be present in #7100 and #7101 and the two byte multiplier in #7102 and #7103. The final four byte product is available in the four addresses, #7104 onwards. The program is merely an extension to the simple algorithm we used for 8 bit numbers. Because of this, we felt a separate flowchart and detailed line by line treatment of the listing were not justified. To understand the program, it is sufficient to point out the following essential differences:

1 Four bytes are needed for the product if overflow is to be avoided. The product is manipulated in registers E,D,C,A (low byte through to high byte). We used the accumulator for the high byte

because **RLA** is faster than **SLA B.** In any case we have used the B register as the loopcounter.
2 Note that two bytes are needed throughout for the multiplier. The HL register pair is used to manipulate this value. Note that the **ADD HL,HL** instruction, in line 180, is used again as a 16 bit shift left. The loop counter, register B, is set to the number of multiplier bits in line 120.
3 If the carry flag is set in line 190, the two byte value for the multiplicand is added to the four byte value of the product. The addition is performed in lines 200 to 280. Note that the HL register pair (multiplier) is temporarily saved on the stack while 16 bit register pair arithmetic is performed.
4 In lines 300 to 320 the high byte of the product in the accumulator is transferred to the, now freed, B register and the product stored in memory using a pair of 16 bit loads.

Signed 16 bit multiplication

Multiplication of numbers where signs must be taken into consideration raises complexity by an order of magnitude. This can be seen by comparing the flowchart shown in Figure 10.2 with the unsigned version shown in Figure 10.1.

To start with, we have to test the sign bits of both multiplicand and the multiplier. The sign rules for multiplication are well known but are worth repeating here:

> Like signs give a positive result.
> Unlike signs give a negative result.

From this, it is fairly easy to work out how these rules can be applied within a program so as to give the correct sign for the result. The following method works by using a spare register or location as a kind of flag.

If the multiplicand is negative, then increment the previously cleared flag.

If the multiplier is negative, then decrement the flag. The result will be as shown in the following table.

Multiplicand	*Multiplier*	*flag value*
+	+	0
−	+	1
+	−	−1
−	−	0

The flag will have a value of zero if the result is positive, and non zero is negative. If either the multiplier and/or the multiplicand is

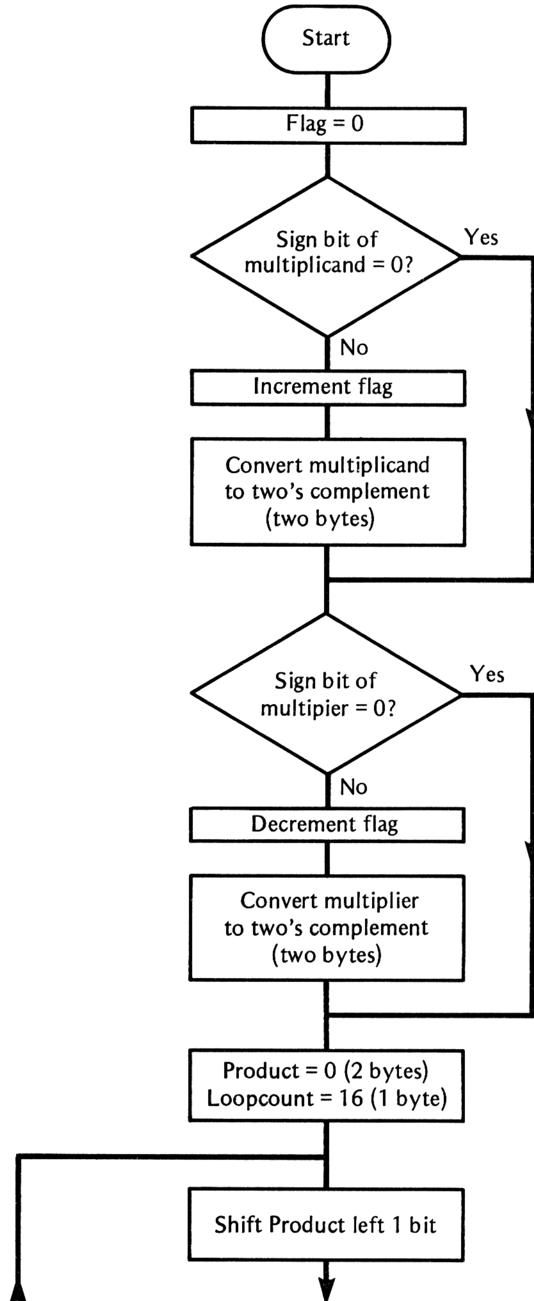
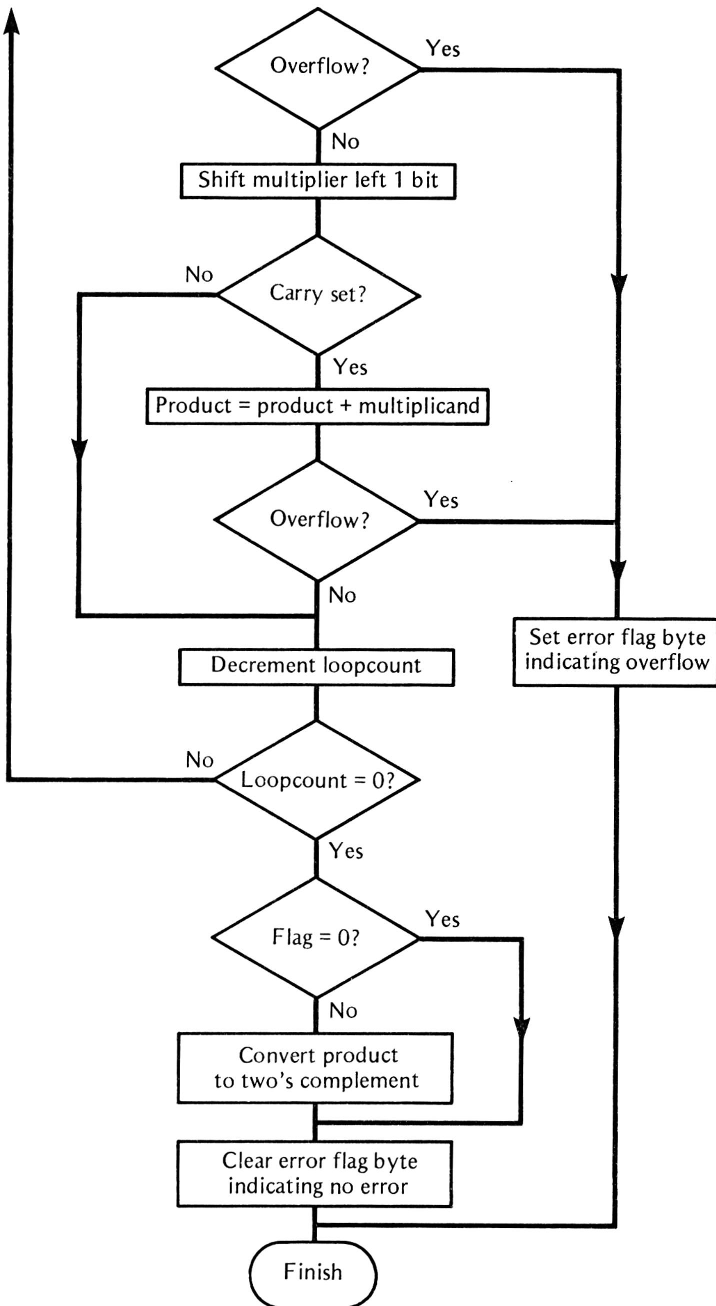


Fig 10.2 Flowchart for signed 16 bit multiplication


Overflow?
Yes
No
Shift multiplier left 1 bit
No
Carry set?
Yes
Product = product + multiplicand
Overflow?
Yes
No
Set error flag byte indicating overflow
Decrement loopcount
No
Loopcount = 0?
Yes
Flag = 0?
Yes
No
Convert product to two's complement
Clear error flag byte indicating no error
Finish

found to be negative, then the two's complement is taken so that all arithmetic is carried out with positive representations. Depending on the value of the flag, the result of the multiplication is left as positive (flag=0) or the two's complement taken if negative (a non zero flag value).

When using signed arithmetic, it is always best to decide on a *standard bit length* for numbers. That is to say, all numbers are represented by 8 bits, 16 bits, 32 bits etc. Attempts to use mixtures can lead to serious problems with large programs. For example, when we were dealing with unsigned integers, we used 16 bit multiplicands and multipliers but a 32 bit result. If, at a later stage, we want to perform further arithmetic on the result, we will need 32 bit multiplication subroutines and so on! Such situations can result in confusion. Confusion nearly always leads to poorly structured programs. For example, Amstrad BASIC treats all integers as 16 bit signed numbers. If we do this, we still need to signal an overflow error if the result is larger than the capacity of 16 bits. All these features are incorporated in Example 10.3, but first, study the flowchart shown in Figure 10.2. (Previous page.)

Note that the information on the sign of the product can be established at the beginning of the flowchart and the flag set accordingly. Multiplication is always performed on positive numbers. Any negative number is converted to its positive form, by taking the two's complement, before multiplication begins. Near the end of the flowchart, after the multiplication process is completed, the flag is examined to decide whether the product is to remain in its positive form or whether the two's complement negative form is to be taken. If overflow is detected, a separate error flag is set to all '1's (#FF). Users of the routine should examine this on return to determine whether or not the product is valid.

The full listing is shown in Example 10.3.

Example 10.3 Signed 16 bit multiplication

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                      10 ;SIGNED 16 BIT MULTIPLICATION
7000                  20 begin:   EQU  #7000
7100                  30 top:     EQU  begin+#100
7100                  40 number:  EQU  top
7102                  50 mult:    EQU  top+2
7104                  60 prod:    EQU  top+4
7106                  70 flag:    EQU  top+6
7000                  80          ORG  begin
7000  0E00            90          LD   C,0
7002  2A0071         100          LD   HL,(number)
7005  CB7C           110          BIT  7,H
7007  280B           120          JR   Z,second
```

```
7009  0C        130          INC  C
700A  EB        140          EX   DE,HL
700B  210000    150          LD   HL,0
700E  A7        160          AND  A
700F  ED52      170          SBC  HL,DE
7011  220071    180          LD   (number),HL
7014  2A0271    190 second:  LD   HL,(mult)
7017  CB7C      200          BIT  7,H
7019  2808      210          JR   Z,start
701B  0D        220          DEC  C
701C  EB        230          EX   DE,HL
701D  210000    240          LD   HL,0
7020  A7        250          AND  A
7021  ED52      260          SBC  HL,DE
7023  110000    270 start:   LD   DE,0
7026  0610      280          LD   B,16
7028  CB23      290 shift:   SLA  E
702A  CB12      300          RL   D
702C  CB7A      310          BIT  7,D
702E  2026      320          JR   NZ,error
7030  29        330          ADD  HL,HL
7031  300E      340          JR   NC,over
7033  3A0071    350          LD   A,(number)
7036  83        360          ADD  A,E
7037  5F        370          LD   E,A
7038  3A0171    380          LD   A,(number+1)
703B  8A        390          ADC  A,D
703C  57        400          LD   D,A
703D  CB7A      410          BIT  7,D
703F  2015      420          JR   NZ,error
7041  10E5      430 over:    DJNZ shift
7043  79        440          LD   A,C
7044  A7        450          AND  A
7045  2806      460          JR   Z,plus
7047  210000    470          LD   HL,0
704A  ED52      480          SBC  HL,DE
704C  EB        490          EX   DE,HL
704D  ED530471  500 plus:    LD   (prod),DE
7051  97        510          SUB  A
7052  320671    520 finish:  LD   (flag),A
7055  C9        530          RET
7056  3EFF      540 error:   LD   A,#FF
7058  18F8      550          JR   finish

Hisoft GENA3.1 Assembler. Page     2.

Pass 2 errors: 00

Table used:    165  from   500
```

How the program works

Before using the program, the two 16 bit numbers to be multiplied must be present in #7100, #7101 and #7102, #7103. The result will be left in #7104, #7105.

Lines 20 to 70: assign labels to locations.

Line 80: forces assembly at #7000.

Lines 90: initialises the sign information flag (register C) to zero.

Lines 100 to 180: check the sign bit of the multiplicand. If positive, a branch is taken to second. If negative, the two's complement of the multiplicand is formed and the sign information flag, register C, is incremented. Note that the two's complement is found by subtracting from zero.
Lines 190 to 260: check the sign bit of the multiplier. If positive, a branch is made to 'start'. If negative, the two's complement is taken and register C is decremented.
Lines 270 to 430: these are similar in form to the unsigned 16 bit multiplication in Example 10.2. However, since we are limiting the product to 16 bits we only need the DE register pair to manipulate the partial product which leads to a more simple listing. Notice the inclusion, in lines 310,320 and 410,420 of the **BIT 7,D** instruction followed by **JR NZ,**error. The reason for these is to check for overflow. Since, in this area of the program, we are working exclusively with positive numbers, any corruption of the partial product's sign bit (bit 7 of the D register) will indicate an overflow, and thus an invalid calculation.
Lines 440 to 490: test the sign information flag (register C). If zero, a branch is made to 'plus'. If non zero, the two's complement of the product is taken.
Line 500: stores the product, currently in the register pair DE, in memory at address #7104 and #7105.
Lines 510 and 520: clears the error flag, indicating that the product is valid and returns.
Lines 540 to 550: sets the error flag to #FF as an overflow signal and returns. Examination of this location can indicate whether overflow has occurred.

Division

Division is the reverse process of multiplication. The four components involved are as follows:

The *dividend* is the number to be divided (the number you start with).
The *divisor* is the number you are dividing by.
The *quotient* is the 'answer' (number of times the divisor goes into the dividend).
The *remainder,* is what is left over, if any.

For example, if we divide 26 by 4, the dividend is 26, the divisor is 4, the quotient is 6 and the remainder is 2.

Alternatively, the remainder can be expressed as a fraction. In our example, the remainder of 2 could be expressed in fractional form as 2/26=1/12. (0.0833.. in decimal form). Since the programs which follow are restricted to integer division, the remainder, if any, is left as a simple integer which may or may not be useful.

Multiplication was achieved by successive shift actions (for examining each bit) and adding, so it was understandable that the product would often be much larger than either of the two component numbers (multiplicand and multiplier). Division, on the other hand, is achieved by shifting and subtracting. This means that the quotient will always be smaller than the dividend, except in the special case when the divisor and the dividend happen to be the same value. Because of this, we need not worry about overflow, neither do we have to allow extra length register combinations for the 'answer'. However, in division, there is one danger lurking beneath the surface which was not present in multiplication. This is the 'division-by zero' error condition. Multiplication by zero is a valid exercise, even if it is useless, but trying to divide by zero is, according to mathematicians (and computers) the worst crime in the book.

8 bit division

The procedure is based on how binary division would be carried out with pencil and paper and is best illustrated in flowchart form (see Figure 10.3). (Overleaf.)

The full listing is shown in Example 10.4, followed by a detailed line by line breakdown.

Example 10.4 Simple 8 bit unsigned division

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                  10 ;UNSIGNED 8 BIT DIVISION
7000              20 begin:  EQU  #7000
7100              30 top:    EQU  begin+#100
7100              40 number: EQU  top
7102              50 div:    EQU  top+2
7103              60 remain: EQU  top+3
7000              70         ORG  begin
7000  2A0071      80         LD   HL,(number)
7003  3A0271      90         LD   A,(div)
7006  4F         100         LD   C,A
7007  3E00       110         LD   A,0
7009  0610       120         LD   B,16
700B  29         130 shift:  ADD  HL,HL
700C  17         140         RLA
700D  B9         150         CP   C
700E  3802       160         JR   C,over
7010  91         170         SUB  C
7011  2C         180         INC  L
```

```
7012  10F7       190 over:   DJNZ shift
7014  320371     200         LD   (remain),A
7017  C9         210         RET

Pass 2 errors: 00

Table used:     94  from   136
```

Example 10.4 is in subroutine form and can be called from any other machine code program. Before calling, the low byte of the dividend must be present in #7100, the high byte in #7101 and the divisor in #7102. The dividend 'number' is loaded and manipulated in the HL register pair. After the subroutine has returned, the quotient is available in the register pair which originally held the dividend, namely HL. The remainder, if any, is present in the accumulator.

Lines 20 to 70: assign labels to locations.

Line 80: forces assembly at #7000.

Lines 90 and 100: the register pair HL is loaded directly from memory with the 16 bit dividend. Register C is loaded, in a somewhat round about fashion, with the corresponding divisor value.

Lines 120 and 130: the accumulator is cleared in line 120, ready for testing the bits shifted out of the msb end of the dividend (HL). The loopcounter, the B register, is initially set to 16, the number of bits in the dividend.

Lines 140 and 150: the **ADD HL,HL** instruction is simply an economical method of performing a 16 bit shift left. The two instructions combine to left shift a bit out of the HL register pair, via the carry status flag, into the accumulator.

Lines 160 and 170: compare the divisor with the accumulator contents. If the carry is found to be set (indicating that the divisor is greater than the accumulator contents) a relative jump to 'over' is made.

Lines 180 and 190: are only executed if the divisor is less than or equal to the accumulator. In this case the divisor is subtracted from the accumulator and the relevant bit of the quotient set to a '1'. A simple dodge is used here to simplify coding. As the Hl register pair is shifted left each time, a zero is fed in from the lsb end of the L register. If we increment the L register, as in line 190, it is equivalent to setting the lsb to a '1'. In this way we can use the same register pair to hold both the dividend, prior to processing, and the quotient which gradually builds up as the loop cycles progress. When the loop is finally vacated, the HL pair will hold only the quotient since all the original dividend bits will have been shifted out.

Line 200: checks if the loopcounter (the B register) has reached zero. If not, it forces another cycle.

Lines 210 and 220: load the quotient and remainder directly into memory.

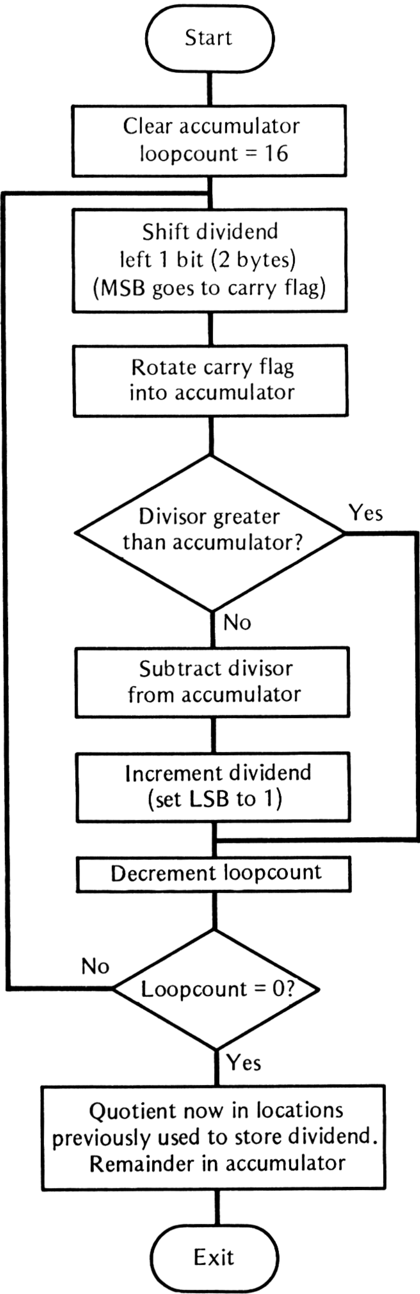


Fig. 10.3 Flowchart for 8 bit division

Unsigned 16 bit division

Example 10.5 assumes that the 16 bit dividend is present in #7100, #7101 and the 16 bit divisor is held in #7102, #7103. On return, the two byte quotient is available in the two locations #7104, #7105. The two byte remainder, if any, is available in the two bytes labelled 'temp' (#7106, #7107).

Example 10.5 Unsigned 16 bit division

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                        10 ;UNSIGNED 16 BIT DIVISION
7000                    20 begin:   EQU  #7000
7100                    30 top:     EQU  begin+#100
7100                    40 number:  EQU  top
7102                    50 div:     EQU  top+2
7104                    60 quot:    EQU  top+4
7106                    70 temp:    EQU  top+6
7000                    80          ORG  begin
7000  210000            90          LD   HL,0
7003  ED4B0071         100          LD   BC,(number)
7007  ED5B0271         110          LD   DE,(div)
700B  78               120          LD   A,B
700C  0610             130          LD   B,16
700E  CB21             140 shift:   SLA  C
7010  17               150          RLA
7011  ED6A             160          ADC  HL,HL
7013  ED52             170          SBC  HL,DE
7015  3803             180          JR   C,skip
7017  0C               190          INC  C
7018  1801             200          JR   over
701A  19               210 skip:    ADD  HL,DE
701B  10F1             220 over:    DJNZ shift
701D  47               230          LD   B,A
701E  ED430471         240          LD   (quot),BC
7022  220671           250          LD   (temp),HL
7025  C9               260          RET

Pass 2 errors: 00

Table used:    114  from    145
```

Example 10.5 is similar to the previous 8 bit example, so a separate flowchart or detailed line by line treatment is scarcely justified. It is sufficient to point out the following differences between the two programs:

1 Two bytes are now needed for the divisor instead of one. The DE register pair is used temporarily to hold and manipulate the divisor.

2 In the 8 bit example, we shifted the dividend bitwise into the accumulator. This time, the HL register pair is used instead of the

accumulator to accommodate 16 bit arithmetic dividend. The **ADC HL,HL** instruction acts as a 16 bit rotate left through carry and is equivalent to an **RL L** instruction followed by an **RL H.** The accumulator is borrowed temporarily as the low byte of the dividend within the loop since **RLA** is more efficient than **RL B,** especially when we consider the loop is executed 16 times. This refinement is set up in line 120 and switched back in line 230.

Note that the carry will always be clear, prior to the **SBC HL,DE** instruction in line 170.

16 bit signed division

When signs of numbers must be allowed for, division subroutines start to become nasty. Earlier we covered the well known sign rules for multiplying two signed numbers together. From these rules, we have to work out a way to apply them to a division program. Fortunately, apart from a change of terms, we can use a similar method to the one described earlier. It employs a spare register, say, the accumulator, as a sign information flag. The accumulator is initially set to contain zero. The method operates as follows:

If the dividend is negative, increment register A.

If the divisor is negative, decrement register A.

The result will be as shown in the following table:

Dividend	*Divisor*	*Accumulator (flag)*
+	+	0
−	+	1
+	−	−1
−	−	0

After the dividend and divisor have been tested, it is clear from the table that the accumulator will have a value of zero if the result is positive and non zero if negative. The above test is carried out before starting the division process. The accumulator is then saved on the stack for later use with a **PUSH AF** instruction. If either the dividend and/or the divisor is found to be negative, the two's complement is taken so that the subsequent division process can be carried out on positive numbers. After the division process the accumulator is retrieved from the stack with a **POP AF** instruction. Depending on the value present in the accumulator, the quotient is left in positive form (if A is zero) or converted to the two's complement negative form (if A is non zero). The flowchart, shown overleaf in Figure 10.4, illustrates the underlying strategy.

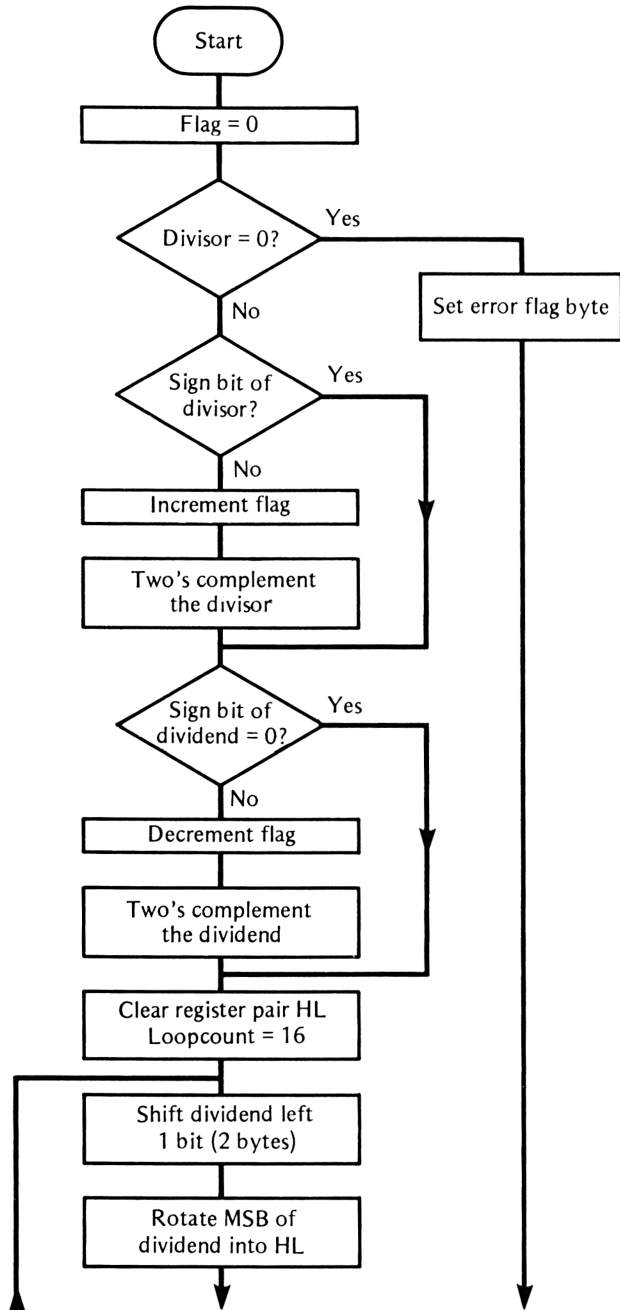
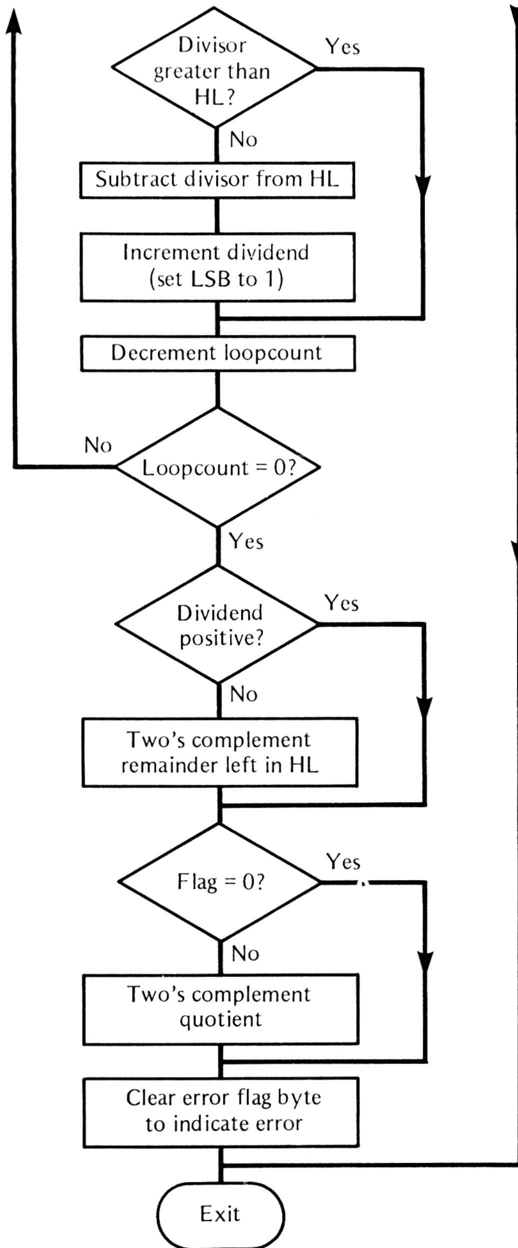


Fig. 10.4 16 bit signed division



As in multiplication, it is always best to decide on a standard bit length when catering for signed integers. In this case, it means that the dividend, divisor and quotient are standardised to a length of 16 bits. It is, of course, possible to arrange division programs for

handling mixed bit lengths but it can lead to confusing coding which is alien to good structure.

The full listing for 16 bit signed division is given in Example 10.6 and can be called as a subroutine from within other machine code programs. Before calling, the two byte dividend must be present in addresses #7100 (low byte) and #7101 (high byte) and the divisor in addresses #7102 (low byte) and #7103 (high byte). On return from the routine, the two byte quotient will be present in #7104 (low byte) and #7105 (high byte). The remainder, if any, will be in #7107 (low byte) and #7108 (high byte). The error flag, location #7106, must be checked before relying on the quotient's validity. A value of zero indicates the quotient is valid whereas a value of #FF indicates a division by zero error.

Example 10.6 Signed 16 bit division

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                     10 ;SIGNED 16 BIT DIVISION
7000                 20 begin:  EQU  #7000
7100                 30 top:    EQU  begin+#100
7100                 40 number: EQU  top
7102                 50 div:    EQU  top+2
7104                 60 quot:   EQU  top+4
7106                 70 flag:   EQU  top+6
7107                 80 remain: EQU  top+7
7000                 90         ORG  begin
7000  ED5B0271      100         LD   DE,(div)
7004  7A            110         LD   A,D
7005  B3            120         OR   E
7006  2859          130         JR   Z,error
7008  97            140         SUB  A
7009  CB7A          150         BIT  7,D
700B  2807          160         JR   Z,second
700D  3C            170         INC  A
700E  210000        180         LD   HL,0
7011  ED52          190         SBC  HL,DE
7013  EB            200         EX   DE,HL
7014  ED4B0071      210 second: LD   BC,(number)
7018  CB78          220         BIT  7,B
701A  2809          230         JR   Z,start
701C  3D            240         DEC  A
701D  210000        250         LD   HL,0
7020  A7            260         AND  A
7021  ED42          270         SBC  HL,BC
7023  4D            280         LD   C,L
7024  44            290         LD   B,H
7025  F5            300 start:  PUSH AF
7026  210000        310         LD   HL,0
7029  78            320         LD   A,B
702A  0610          330         LD   B,16
702C  CB21          340 shift:  SLA  C
```

```
702E  17          350          RLA
702F  ED6A        360          ADC   HL,HL
7031  ED52        370          SBC   HL,DE
7033  3803        380          JR    C,skip
7035  0C          390          INC   C
7036  1801        400          JR    over
7038  19          410 skip:    ADD   HL,DE
7039  10F1        420 over:    DJNZ  shift
703B  47          430          LD    B,A
703C  3A0171      440          LD    A,(number+1)
703F  CB7F        450          BIT   7,A
7041  2807        460          JR    Z,rempos
7043  EB          470          EX    DE,HL
7044  210000      480          LD    HL,0
7047  A7          490          AND   A
7048  ED52        500          SBC   HL,DE
704A  220771      510 rempos:  LD    (remain),HL
704D  F1          520          POP   AF
704E  A7          530          AND   A
704F  2807        540          JR    Z,plus
7051  210000      550          LD    HL,0
7054  ED42        560          SBC   HL,BC

Hisoft GENA3.1 Assembler. Page     2.

7056  4D          570          LD    C,L
7057  44          580          LD    B,H
7058  ED430471    590 plus:    LD    (quot),BC
705C  97          600          SUB   A
705D  320671      610 finish:  LD    (flag),A
7060  C9          620          RET
7061  3EFF        630 error:   LD    A,#FF
7063  18F8        640          JR    finish

Pass 2 errors: 00

Table used:    201  from   500
```

The line by line break down is as follows:

Lines 20 to 80: assign labels to locations.

Line 90: forces assembly at address #7000.

Lines 100 to 130: check if the divisor is zero and, if so, causes a branch to 'error'.

Line 140: initialises the sign information flag (accumulator) to zero. Note that **SUB A** is a quick method of clearing the accumulator.

Lines 150 to 200: check sign bit of divisor (bit 7 of register D). If the divisor is positive, a branch is made to 'second'. If negative, the two's complement of the divisor is formed and the accumulator incremented. There is no need to clear the carry status flag before the **SBC HL,DE** instruction since it will have been cleared by the **SUB A** instruction in line 140.

Lines 210 to 290: check the sign bit of the dividend (bit 7 of register B). If positive, a branch is made to 'start'. If negative, the two's complement of the dividend is formed and the accumulator decremented.

Line 300: pushes the sign information flag (accumulator) on the stack for later use.
Lines 310 to 420: perform the division operation. Because division is carried out on positive values for dividend and divisor, the coding is similar to lines 90 to 130 in Example 10.5 which dealt with unsigned 16 bit division.
Lines 440 to 500: test the sign of the dividend. If positive then the remainder will be left in positive form. If negative, the two's complement of the remainder is taken.
Line 510: the remainder, left in the HL register pair, is stored directly in memory starting at the location labelled 'remain'.
Line 520: the sign information data, previously saved on the stack, is restored to the accumulator.
Line 530: updates the status flags, without altering the accumulator contents, by using **AND A.** (The **POP** instruction does not update any flags.)
Lines 540 to 580: tests the accumulator for zero. If zero, a branch is made to 'plus'. If non zero, the two's complement of the quotient is taken.
Line 590: stores the quotient, currently in the register pair BC, in address #7104 and #7105.
Lines 600 and 620: clear the error flag, indicating the division is valid before returning.
Lines 630 and 640: set the error flag to #FF as a division by zero signal before returning. Later interrogation of this location, #7106, can indicate whether or not division by zero has occurred.

Summary

1 The multiplicand is one number, the multiplier is the other and the result is the product.
2 Like signs give positive, unlike signs give a negative result.
3 Using a constant bit length simplifies a multiplication program.
4 Multiplication is carried out by a combination of left shift and addition.
5 Signed multiplication is best carried out on positive numbers, leaving the sign of the product until last.
6 The four components of division are the dividend, the divisor, the quotient and the remainder.
7 Division is carried out by a combination of left shift and subtraction.

Input and output 11

Input and output routines, which enable the user to enter and display string and numerical data in a user friendly form, are now described.

Nearly all data input at a computer keyboard is entered as a character string and temporarily stored in a section of memory called a buffer. Whether the characters entered into the buffer are left in string form or converted to numeric data and stored elsewhere is the task of an input routine. It follows that a string input routine is a common building brick of all keyboard input, whether string or numerical.

String input routines

Although it is possible to delve into the BASIC interpreter to find a ready made input routine, it is far more rewarding to construct your own. In any case you would have to conform to the general purpose philosophy adopted by the original software design team. For example, we may not necessarily require strings to be limited to 255 characters. However, a few pointers can be gained by examining the following attributes of a typical BASIC string input routine:

(a) An input *prompt* character such as a cursor or question mark.
(b) A method of reflecting the characters input to the screen.
(c) A method of back deleting characters, both from the screen and the keyboard buffer, in the event of mistyped characters. The DELETE key is normally used for this purpose.
(d) A physical and audible limit on back deletion, no further than the first character input.
(e) A similar limit on the maximum allowed number of characters. A limit of 255 is normally allowable in BASIC but this could be increased or decreased if we write our own input routines.
(f) The receipt of an ENTER key code terminates the string input.

A flow chart incorporating all these factors is shown in Figure 11.1.

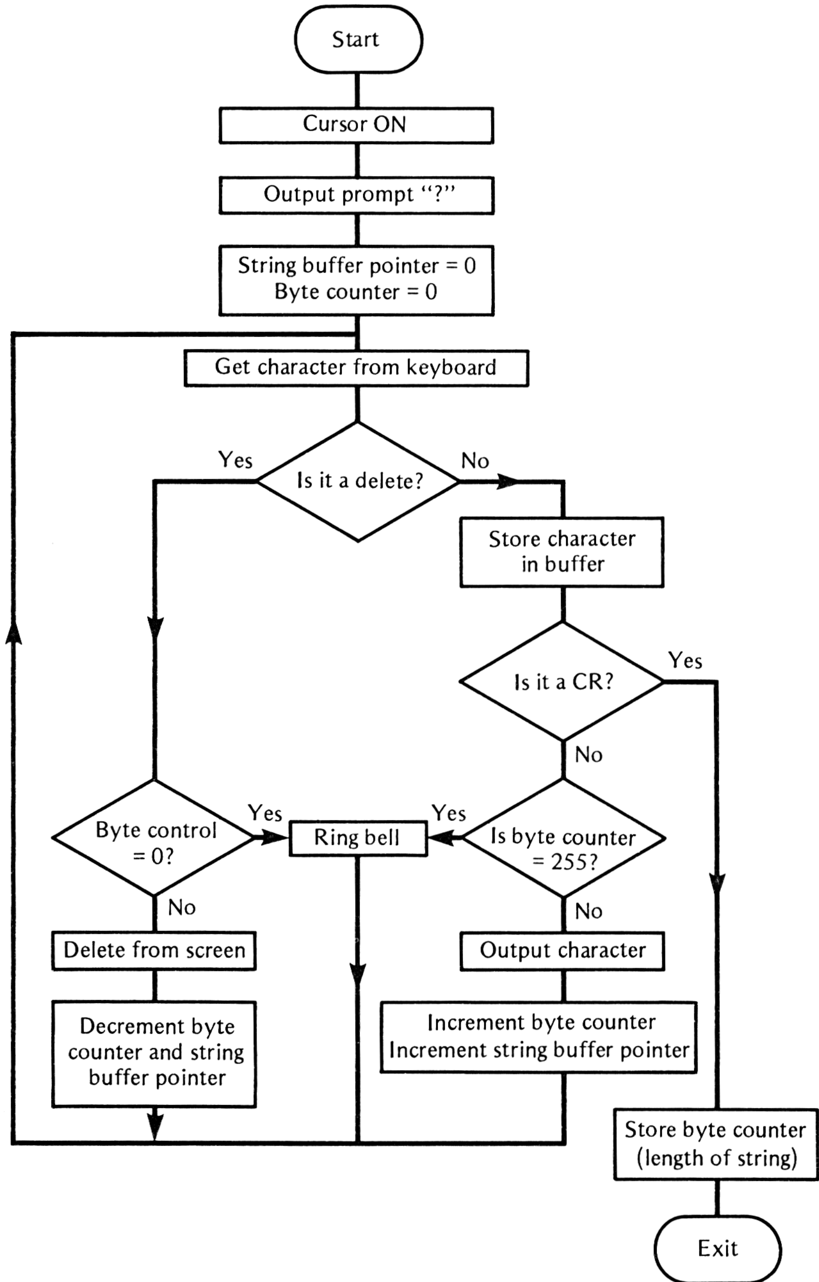


Fig 11.1 Flow chart of a string input routine

The flow chart can be coded fairly efficiently by making use of a few text VDU control codes and firmware routines. Program 11.1 is the result. At various points in the program breakdown, it may be necessary to refer back to Chapter 7 which deals with commonly used firmware routines.

Program 11.1 String input routine

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                        10 ;STRING INPUT ROUTINE
7000                    20 begin:   EQU  #7000
7100                    30 top:     EQU  begin+#100
7100                    40 length:  EQU  top
7101                    50 string:  EQU  top+1
BB18                    60 WAITKY:  EQU  #BB18
BB5A                    70 OUTPUT:  EQU  #BB5A
BB81                    80 CURON:   EQU  #BB81
7000                    90          ORG  begin
7000   CD81BB          100          CALL CURON
7003   3E3F            110          LD   A,#3F
7005   CD5ABB          120          CALL OUTPUT
7008   210171          130          LD   HL,string
700B   0600            140          LD   B,0
700D   CD18BB          150 loop:    CALL WAITKY
7010   FE7F            160          CP   #7F
7012   2019            170          JR   NZ,accept
7014   78              180          LD   A,B
7015   A7              190          AND  A
7016   280E            200          JR   Z,bell
7018   3E08            210          LD   A,8
701A   CD5ABB          220          CALL OUTPUT
701D   3E10            230          LD   A,16
701F   CD5ABB          240          CALL OUTPUT
7022   05              250          DEC  B
7023   2B              260          DEC  HL
7024   18E7            270          JR   loop
7026   3E07            280 bell:    LD   A,7
7028   CD5ABB          290          CALL OUTPUT
702B   18E0            300          JR   loop
702D   77              310 accept:  LD   (HL),A
702E   FE0D            320          CP   #0D
7030   280E            330          JR   Z,exit
7032   4F              340          LD   C,A
7033   78              350          LD   A,B
7034   FEFF            360          CP   #FF
7036   28EE            370          JR   Z,bell
7038   79              380          LD   A,C
7039   CD5ABB          390          CALL OUTPUT
703C   04              400          INC  B
703D   23              410          INC  HL
703E   18CD            420          JR   loop
7040   78              430 exit:    LD   A,B
```

```
7041  320071      440         LD    (length),A
7044  C9          450         RET

Pass 2 errors: 00

Table used:    145  from   172
```

Breakdown of Program 11.1

The purpose of Program 11.1 is to obtain string characters from the user, via the keyboard, reflect them onto the screen and store them in a buffer area along with a string length byte. All the features mentioned in (a) to (f) above are incorporated.

Lines 20 to 50: assign labels to the locations used. The string itself is stored in a buffer, up to 255 bytes long, starting at address #7101. This can of course be altered to suit your requirements. The length of the string is placed in location #7100.

Lines 60 to 80: assign labels to the firmware routines used. Chapter 7 gives more details of these. The labels used in lines 60 to 80 are abbreviations for **KM WAIT KEY, TXT OUTPUT,** and TXT CUR ON respectively.

Line 90: is the **ORG** assembler directive which forces assembly at address #7000.

Line 100: calls the **KM CUR ON** firmware routine which places the cursor on the screen.

Lines 110 to 120: output to the screen a '?' character to act as an input prompt.

Lines 130 to 140: initialise the string buffer pointer, register pair HL, and set the byte counter, register B, to zero.

Line 150: is the start of the main input loop extending to line 420. A character, entered at the keyboard, is read into the accumulator.

Lines 160 to 170: The accumulator is checked for the delete key code #7F. If the code is not #7F then a branch is made to 'accept'.

Lines 180 to 200: check the byte counter for zero. If zero a branch is made to 'bell' as a warning.

Lines 210 to 220: output text VDU code 8 named BS which makes the current cursor position legal. They then move the cursor left one character.

Lines 230 to 240: output text VDU code 16, named DLE, which makes the current cursor position legal and clears it to the current paper ink. Note that lines 210 to 240 in conjunction perform a single backspace delete operation on the screen.

Lines 250 to 260: decrement both the byte counter and the string buffer pointer. This is so the deleted character on the screen above is also effectively cleared from the buffer area.

Line 270: branches back to 'loop' for the next character from the keyboard.
Lines 280 to 300: output text VDU code 7 named BEL which makes a short audible beep. The program then branches back to 'loop' for the next character.
Line 310: stores the character, present in the accumulator, in the current buffer position using implied addressing.
Lines 320 to 330: check if the current character is a carriage return ('ENTER' key code #0D) and if so terminate the routine by a branch to 'exit'.
Line 340: temporarily stores the accumulator contents in register C so that the accumulator can be used for other purposes. The contents are later replaced in the accumulator in line 380.
Lines 350 to 370: check if the byte counter has reached the set limit of #FF. If so, a branch is made to 'bell' as a warning. The limit can be reduced to less than #FF if required. If strings longer than 255 characters are needed then a register pair, say DE, can be used as the byte counter.
Lines 380 to 390: output the current character on to the screen.
Lines 400 to 420: increment both the byte counter and the string buffer pointer before a branch is made back to 'loop' for the next character.
Lines 430 to 450: store the byte counter or string length byte in memory and returns control to the calling program.

Notes
Once the string is present in the buffer area it can be moved, including its specific length byte, to any part of memory for storage by the main program. Alternatively we could append a #0D character to delimit different strings in memory. When handling a lot of strings access tables would be important. An array or contiguous list can be built up by storing the actual strings sequentially in memory and setting up an *access table* consisting of three byte groups. The first byte could be string *length* and the second and third bytes the string *address*, low byte and high byte respectively. A simple machine code filing program could be set up in this way.

String output

This is much simpler to perform and is more or less the reverse of the previous operation. We assume a string has been moved to an arbitrary buffer area, starting at say the location labelled 'string' (#7101), and it is required to print it out on the screen. The length of the string is assumed to be present in the location labelled 'length' (#7100). The simple process hardly justifies a flow chart. Instead, just an example listing is given.

Program 11.2 String output routine

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                      10 ;STRING OUTPUT ROUTINE
7000                  20 begin:  EQU  #7000
7100                  30 top:    EQU  begin+#100
7100                  40 length: EQU  top
7101                  50 string: EQU  top+1
BB5A                  60 OUTPUT: EQU  #BB5A
7000                  70         ORG  begin
7000  210171          80         LD   HL,string
7003  3A0071          90         LD   A,(length)
7006  47             100         LD   B,A
7007  7E             110 out:    LD   A,(HL)
7008  CD5ABB         120         CALL OUTPUT
700B  23             130         INC  HL
700C  10F9           140         DJNZ out
700E  C9             150         RET

Pass 2 errors: 00

Table used:    84  from   128
```

Breakdown of Program 11.2

Lines 20 to 70: set up the labelled locations as mentioned above.
Line 80: loads the register pair HL with the start address of the chosen string buffer area.
Lines 90 to 100: load the byte counter, B register, with the string length byte.
Lines 110 to 140: output the string characters in a loop fashion using implied addressing.
Line 150: returns control to the calling program.

Text output

Literal text output is a common requirement. In BASIC we simply write a line of the type:

PRINT 'The elephants are coming'

in the body of the program. When in assembly language it is often convenient to place all literal text strings at the *foot* of a program as you would DATA statements in BASIC. With assembly language, we have the added advantage of using labels to distinguish between different strings or messages.

With the HiSoft DEVPAC assembler, literal strings can be assembled and stored in memory with the DEFM assembler directive. For example:

DEFM 'The elephants are coming'

will, on assembly, place the ASCII codes for each character of the above string in memory. Using labels to distinguish different messages we could well have:

first: DEFM 'The elephants are coming'
second: DEFM 'the elephants have arrived'
third: DEFM 'My leg is broken'
fourth: DEFM 'I hope the elephants will go

A method is needed to distinguish a particular message in the list for printing to the screen. Program 11.3 is an example of how this can be done in practice and is split into three modules. The first group of instructions is simply an example of how to specify which message to print and calls the 'print' subroutine, lines 160 to 280. In a large program, you would only print the selected message. The second module prints out the specified message using implied addressing. The third module is the list of labelled DEFM assembler directives containing the messages themselves. Notice the termination characters used at the end of the strings. They each have a specific meaning within the print subroutine and must be included. The ';' character terminator is used to specify 'the follow on' of text. That is to say no line feeds or spaces are placed after the message. In fact the ';' character is used in much the same way as it is in BASIC PRINT statements with the exception that it is placed within, rather than outside, the quotes. The ']' character is the one chosen to signal a new line. Other characters can be used for this purpose, by changing lines 170 and 190.

Program 11.3 Text output routine

```
Hisoft GENA3.1 Assembler. Page      1.

Pass 1 errors: 00

                    10 ;TEXT OUTPUT ROUTINE
7000                20 begin:   EQU  #7000
BB5A                30 OUTPUT:  EQU  #BB5A
7000                40          ORG  begin
                    50 ;
7000  213370        60          LD   HL,first
7003  CD1970        70          CALL print
7006  214570        80          LD   HL,second
7009  CD1970        90          CALL print
700C  215770       100          LD   HL,third
700F  CD1970       110          CALL print
7012  216970       120          LD   HL,fourth
7015  CD1970       130          CALL print
7018  C9           140          RET
                   150 ;
7019  7E           160 print:   LD   A,(HL)
701A  FE3B         170          CP   ";"
```

```
7O1C  2814        180            JR    Z,exit
7O1E  FE5D        190            CP    "]"
7020  2806        200            JR    Z,newlin
7022  CD5ABB      210            CALL  OUTPUT
7025  23          220            INC   HL
7026  18F1        230            JR    print
7028  3E0D        240 newlin:    LD    A,#0D
702A  CD5ABB      250            CALL  OUTPUT
702D  3E0A        260            LD    A,#0A
702F  CD5ABB      270            CALL  OUTPUT
7032  C9          280 exit:      RET
                  290 ;
7033  54484953    300 first:     DEFM  "THIS IS MESSAGE 1;"
7045  54484953    310 second:    DEFM  "THIS IS MESSAGE 2]"
7057  54484953    320 third:     DEFM  "THIS IS MESSAGE 3]"
7069  54484953    330 fourth:    DEFM  "THIS IS MESSAGE 4]"

Pass 2 errors: 00

Table used:    124  from    163
```

Breakdown of Program 11.3

Line 60: loads the HL register pair with the address of the message labelled 'first' at the foot of the program.
Line 70: calls the print subroutine which displays the message 'first' if HL is loaded as above.
Lines 80 to 130: similarly print out the other messsages by loading HL with the appropriate address and calling the print routine.
Line 160: starts the print subroutine. Character codes are loaded one at a time into the accumulator in a loopwise fashion using implied addressing.
Lines 170 to 180: force a branch to 'exit' if a ';' character code is detected in the accumulator.
Lines 190 to 200: force a branch to 'newlin' if a ']' character is detected in the accumulator.
Line 210: uses the firmware routine **TXT OUTPUT** to place the character on the screen at the current cursor position.
Lines 220 to 230: increment the HL string character pointer and a branch back to 'print' is effected to get the next character of the message.
Lines 240 to 270: output, via the firmware routine **TXT OUTPUT,** the text VDU codes #0D and #0A. The #0D code, CR, brings the cursor back to the beginning of current line and the code #0A, LF, is a line feed. This section is only executed if the string is terminated with a ']' character, detected in line 200.
Lines 300 to 330: are the labelled text messages in string form. Note that because of the ']' character in line 300 the screen output will be as follows:

THIS IS MESSAGE 1THIS IS MESSAGE 2
THIS IS MESSAGE 3
THIS IS MESSAGE 4

No line feed is specified after the first message.

Decimal and hexadecimal input routines

It is a comparatively easy task to input ASCII characters and store them as strings in memory locations. When we wish to enter actual numbers it becomes a little more difficult. The method normally adopted is to use a string input routine to get the number, in ASCII character form, into a temporary buffer area. The buffer contents can then be validated and converted to the binary equivalent at a later stage. Throughout the examples given, we will convert all input to 16 bit signed binary. The routine could easily be extended along the same lines to handle 32 bit or more signed integers if required.

Decimal input

The most user friendly way to input numbers at the keyboard is in decimal form. Assuming the characters have been entered at the keyboard and placed in a buffer area, it is then necessary to convert the ASCII codes, representing the number, into signed 16 bit binary. The algorithm which follows is fairly simple:

(a) Clear a 16 bit register pair, say, DE.
(b) Multiply the contents of the DE register pair by 10.
(c) Starting from the most significant end, convert next decimal digit from its ASCII code form to binary by subtracting hex 30.
(d) Add the above result, in (c), to the DE register pair.
(e) Branch back to (b) until all characters are processed.
(f) The resulting number in DE is the 16 bit binary representation of the ASCII string.

For example, if the characters 5694 are to be converted:

Cycle	*ASCII*	*ASCII-#30*	*DE contents*
1	#35	5	(0×10)+5=5
2	#36	6	(5×10)+6=56
3	#39	9	(56×10)+9=569
4	#34	4	(569×10)+4=5694

The above, although workable, is an oversimplification; we also need to take into account such factors as validation of characters,

sign and overflow. A worthwhile decimal input routine should have at least the following extras:

1 Detect and process a leading '–' sign.
2 Handle an optional leading '+' sign.
3 Detect overflow conditions and set an error flag accordingly. With 16 bit signed binary the acceptable range will be −32768 to +32767.
4 Detect invalid characters entered and set an appropriate error flag. The characters 0123456789−+ are the only ones acceptable.

Program 11.4 is an example which incorporates the above features. The 16 bit binary number will be stored in locations #7100 (low byte) and #7101 (high byte). To test the program these locations can be PEEKed from BASIC.

Program 11.4 Decimal input routine (16 bit)

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                        10 ;DECIMAL INPUT ROUTINE
                        20 ;(decimal to signed 16 bit binary)
7000                    30 begin:   EQU   #7000
7100                    40 top:     EQU   begin+#100
7100                    50 number:  EQU   top
7102                    60 flag:    EQU   top+2
7103                    70 stackp:  EQU   top+3
7105                    80 string:  EQU   top+5
BB18                    90 WAITKY:  EQU   #BB18
BB5A                   100 OUTPUT:  EQU   #BB5A
BB81                   110 CURON:   EQU   #BB81
7000                   120          ORG   begin
7000   ED730371        130          LD    (stackp),SP
7004   CD81BB          140          CALL  CURON
7007   3E3F            150          LD    A,#3F
7009   CD5ABB          160          CALL  OUTPUT
700C   210571          170          LD    HL,string
700F   0600            180          LD    B,0
7011   CD18BB          190 loop:    CALL  WAITKY
7014   FE7F            200          CP    #7F
7016   2019            210          JR    NZ,accept
7018   78              220          LD    A,B
7019   A7              230          AND   A
701A   280E            240          JR    Z,bell
701C   3E08            250          LD    A,8
701E   CD5ABB          260          CALL  OUTPUT
7021   3E10            270          LD    A,16
7023   CD5ABB          280          CALL  OUTPUT
7026   05              290          DEC   B
7027   2B              300          DEC   HL
7028   18E7            310          JR    loop
702A   3E07            320 bell:    LD    A,7
```

```
702C  CD5ABB    330          CALL  OUTPUT
702F  18E0      340          JR    loop
7031  77        350 accept:  LD    (HL),A
7032  FE0D      360          CP    #0D
7034  280E      370          JR    Z,exit
7036  4F        380          LD    C,A
7037  78        390          LD    A,B
7038  FEFF      400          CP    #FF
703A  28EE      410          JR    Z,bell
703C  79        420          LD    A,C
703D  CD5ABB    430          CALL  OUTPUT
7040  04        440          INC   B
7041  23        450          INC   HL
7042  18CD      460          JR    loop
7044  0E00      470 exit:    LD    C,0
7046  110000    480          LD    DE,0
7049  210571    490          LD    HL,string
704C  7E        500          LD    A,(HL)
704D  FE2B      510          CP    "+"
704F  2807      520          JR    Z,skip
7051  FE2D      530          CP    "-"
7053  2004      540          JR    NZ,change
7055  3E80      550          LD    A,#80
7057  4F        560          LD    C,A

Hisoft GENA3.1 Assembler. Page     2.

7058  23        570 skip:    INC   HL
7059  E5        580 change:  PUSH  HL
705A  7E        590          LD    A,(HL)
705B  FE0D      600          CP    #0D
705D  282E      610          JR    Z,done
705F  D630      620          SUB   #30
7061  3846      630          JR    C,error
7063  FE0A      640          CP    10
7065  3042      650          JR    NC,error
7067  F5        660          PUSH  AF
7068  210000    670          LD    HL,0
706B  3E0A      680          LD    A,10
706D  0608      690          LD    B,8
706F  29        700 shift:   ADD   HL,HL
7070  CB7C      710          BIT   7,H
7072  2031      720          JR    NZ,ovflow
7074  17        730          RLA
7075  3005      740          JR    NC,over
7077  19        750          ADD   HL,DE
7078  CB7C      760          BIT   7,H
707A  2029      770          JR    NZ,ovflow
707C  10F1      780 over:    DJNZ  shift
707E  F1        790          POP   AF
707F  85        800          ADD   A,L
7080  5F        810          LD    E,A
7081  3E00      820          LD    A,0
7083  8C        830          ADC   A,H
7084  57        840          LD    D,A
7085  CB7A      850          BIT   7,D
7087  201C      860          JR    NZ,ovflow
```

```
7089  E1          870          POP  HL
708A  23          880          INC  HL
708B  18CC        890          JR   change
708D  CB79        900 done:    BIT  7,C
708F  2807        910          JR   Z,bypass
7091  210000      920          LD   HL,0
7094  A7          930          AND  A
7095  ED52        940          SBC  HL,DE
7097  EB          950          EX   DE,HL
7098  ED530071    960 bypass:  LD   (number),DE
709C  97          970          SUB  A
709D  320271      980 finish:  LD   (flag),A
70A0  ED7B0371    990          LD   SP,(stackp)
70A4  C9         1000          RET
70A5  3EFF       1010 ovflow:  LD   A,#FF
70A7  18F4       1020          JR   finish
70A9  3E01       1030 error:   LD   A,1
70AB  18F0       1040          JR   finish

Pass 2 errors: 00

Table used:   278  from   500
```

Breakdown of Program 11.4

Lines 20 to 120: set up the familiar label assignments and assembler directives.
Line 130: saves the current value of the stack pointer in memory. This will be reinstated later in line 990.
Lines 140 to 460: are identical to the string input routine Program 11.1.
Lines 470 to 490: initialise the registers. The C register is used to store the sign information of the number input which is set to zero for positive and #80 for negative. The DE register pair, set initially to zero, is used to contain the 16 bit binary number as it builds up. The HL register pair is loaded with the address of the temporary buffer area where the input string is stored.
Lines 500 to 520: load the first string character code into the accumulator. If it is a '+' sign then a branch to 'skip' is made.
Lines 530 to 540: if it is not a '−' sign then a branch to 'change' is forced.
Lines 550 to 560: set register C to #80 as a flag for later use.
Line 570: increments the data pointer, HL, to the first numerical character.
Line 580: pushes the data pointer on the stack so as to free HL for other duties.
Lines 590 to 610: check if the current string character, loaded into the accumulator, is a carriage return (#0D). If so the loop is terminated by a branch to 'done'.
Lines 620 to 630: converts the ASCII code of the decimal digit to binary by subtracting 30 hex. If the carry is set after the subtraction

then the ASCII code must have been less than #30 so a branch to 'error' is performed.
Lines 640 to 650: check if the accumulator contents is less than 10. If not a branch to 'error', indicating an invalid character, is performed.
Line 660: saves the accumulator, containing the current decimal digit, on the stack in order to free the accumulator for other tasks.
Lines 670 to 780: contain a simple multiplication by 10 algorithm of the type discussed in chapter 10. Note that line 700 is equivalent to a 16 bit shift left. The DE register pair, holding the building 16 bit number, is multiplied by 10. The resulting product is in the HL register pair. Twice, the sign bit of the product is checked. If set to '1' then overflow is detected and a branch to 'ovflow' is made.
Lines 790 to 840: restore the decimal digit, pushed on the stack in line 660. It is then added to the register pair, HL, and stored back in the register pair DE for the next cycle of the loop.
Lines 840 to 860: check for overflow after the previous addition.
Lines 870 to 890: restore the current data pointer, previously saved on the stack in line 580, increment it to the next string character and force a branch back to 'change' to repeat the process starting at line 580.
Lines 900 to 950: after exit from the loop, the sign bit of the C register is checked. This was used earlier to save the sign information of the number input. If the sign bit (bit 7) is zero then the result is positive and a branch to 'bypass' is performed. If set to a '1' then the two's complement of DE is taken as the negative representation.
Lines 960 to 980: store the 16 bit number, present in the DE register pair, into memory and clear the location #7102 labelled 'flag' to zero. This indicates an error free execution.
Line 990: restores the stack pointer contents which prevailed at the start of the program. This is a precaution taken against possible error conditions, in which case useless information may be left on the stack.
Lines 1010 to 1020: set the location 'flag' to #FF indicating overflow.
Lines 1030 to 1040: set the location 'flag' to '1' indicating invalid input characters. The error flag can be interrogated on return and the input validity checked.

Hexadecimal input

In some programs a hexadecimal input routine may be required. The method used in Program 11.5 is similar to that adopted in the previous example. The algorithm is outlined below:

(a) Clear a 16 bit register pair, say, DE.
(b) This time, multiply the contents of the DE register pair by 16.
(c) Starting from the most significant end, convert next hexadecimal digit from its ASCII code form to binary. This is a little more complex than in the decimal example.

(d) Add the above result, in (c), to the DE register pair.
(e) Branch back to (b) until all characters are processed.
(f) The resulting number in DE is the 16 bit binary representation of the ASCII string.

For example, if the characters 1AD3 are to be converted.

Cycle	*ASCII*	*Hex digit*	*DE contents*
1	#31	1	(0×16)+1=1
2	#41	A	(1×16)+A=1A
3	#44	D	(1A×16)+D=1AD
4	#33	3	(1AD×16)+3=1AD3

The conversion from ASCII to decimal is simple. Just subtract 30 hex. The conversion of ASCII to hexadecimal digits is more complex due to the gap in the ASCII codes between 9 and A. See the program 11.5 breakdown for one method of performing this. We need not bother about sign since this will be implied from the hex digits input. For example – 32768 will be entered as &8000 and + 32767 will be entered as &7FFF. Again overflow and character validity should be checked.

A 16 bit hexadecimal input routine should perform at least the following actions:

1 Handle an optional leading '&' prefix.
2 Detect overflow conditions and set an error flag accordingly. With 16 bit signed binary the acceptable range will be −32768 (8000 hex) to +32767 (7FFF hex).
3 Detect invalid characters entered and set an error flag accordingly. The characters 0123456789ABCDEF& are the only ones acceptable.

Program 11.5 is an example which incorporates all the above features. The 16 bit binary number will be stored in locations #7100 (low byte) and #7101 (high byte). To test the program these locations can be PEEKed from BASIC.

Program 11.5 Hexadecimal input routine (16 bit)

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                        10 ;HEXADECIMAL INPUT ROUTINE
                        20 ;(16 bit with validation)
7000                    30 begin:  EQU  #7000
7100                    40 top:    EQU  begin+#100
7100                    50 number: EQU  top
```

```
7102                60 flag:    EQU  top+2
7103                70 string:  EQU  top+3
BB18                80 WAITKY:  EQU  #BB18
BB5A                90 OUTPUT:  EQU  #BB5A
BB81               100 CURON:   EQU  #BB81
7000               110          ORG  begin
7000  CD81BB       120          CALL CURON
7003  3E3F         130          LD   A,#3F
7005  CD5ABB       140          CALL OUTPUT
7008  210371       150          LD   HL,string
700B  0600         160          LD   B,0
700D  CD18BB       170 loop:    CALL WAITKY
7010  FE7F         180          CP   #7F
7012  2019         190          JR   NZ,accept
7014  78           200          LD   A,B
7015  A7           210          AND  A
7016  280E         220          JR   Z,bell
7018  3E08         230          LD   A,8
701A  CD5ABB       240          CALL OUTPUT
701D  3E10         250          LD   A,16
701F  CD5ABB       260          CALL OUTPUT
7022  05           270          DEC  B
7023  2B           280          DEC  HL
7024  18E7         290          JR   loop
7026  3E07         300 bell:    LD   A,7
7028  CD5ABB       310          CALL OUTPUT
702B  18E0         320          JR   loop
702D  77           330 accept:  LD   (HL),A
702E  FE0D         340          CP   #0D
7030  280E         350          JR   Z,exit
7032  4F           360          LD   C,A
7033  78           370          LD   A,B
7034  FEFF         380          CP   #FF
7036  28EE         390          JR   Z,bell
7038  79           400          LD   A,C
7039  CD5ABB       410          CALL OUTPUT
703C  04           420          INC  B
703D  23           430          INC  HL
703E  18CD         440          JR   loop
7040  110000       450 exit:    LD   DE,0
7043  210371       460          LD   HL,string
7046  7E           470          LD   A,(HL)
7047  FE26         480          CP   "&"
7049  2001         490          JR   NZ,getchr
704B  23           500          INC  HL
704C  7E           510 getchr:  LD   A,(HL)
704D  FE0D         520          CP   #0D
704F  2823         530          JR   Z,done
7051  FE47         540          CP   #47
7053  302C         550          JR   NC,error
7055  D630         560          SUB  #30

Hisoft GENA3.1 Assembler. Page     2.

7057  3828         570          JR   C,error
7059  FE0A         580          CP   10
705B  3806         590          JR   C,over
```

```
705D  D607       600          SUB  7
705F  FE0A       610          CP   10
7061  381E       620          JR   C,error
7063  0604       630 over:    LD   B,4
7065  CB23       640 shift:   SLA  E
7067  CB12       650          RL   D
7069  3812       660          JR   C,ovflow
706B  10FB       670          DJNZ shift
706D  83         680          ADD  A,E
706E  380D       690          JR   C,ovflow
7070  5F         700          LD   E,A
7071  23         710          INC  HL
7072  18D8       720          JR   getchr
7074  ED530071   730 done:    LD   (number),DE
7078  97         740          SUB  A
7079  320271     750 finish:  LD   (flag),A
707C  C9         760          RET
707D  3EFF       770 ovflow:  LD   A,#FF
707F  18F8       780          JR   finish
7081  3E01       790 error:   LD   A,1
7083  18F4       800          JR   finish

Pass 2 errors: 00

Table used:    241   from    500
```

Breakdown of Program 11.5

Lines 120 to 440: are identical to the string input routine Program 11.1
Lines 450 to 460: perform initialisation of registers. The DE register pair, set initially to zero, is used to contain the 16 bit binary number as it is built up. The HL register pair is loaded with the address of the temporary buffer area where the input string is stored.
Lines 470 to 490: load the first string character code into the accumulator. If it is '&' then a branch to 'getchr' is performed.
Line 500: increments the data pointer, HL, to the first hexadecimal character.
Lines 510 to 530: check if the current string character, loaded into the accumulator, is a carriage return (#0D). If so the loop is terminated by a branch to 'done'.
Lines 540 to 550: check that the ASCII code of the current character is less than #47. If not a branch to 'error' indicates an invalid character.
Lines 560 to 570: converts the ASCII code of the hexadecimal digit to binary by first subtracting #30. If, after the subtraction, the carry is set then the ASCII code must have been less than #30 so a branch to 'error' is performed.
Lines 580 to 590: check if the accumulator contents is less than 10. If so a branch to 'over' is performed, indicating that the ASCII code was valid and originally less than that of the character 9.
Line 600: subtracts 7 from the accumulator to compensate for the gap between the ASCII codes mentioned earlier.

Lines 610 to 620: again check if the accumulator contents is less than 10. If so then a branch to 'error' is forced because the original ASCII code must have been one of the characters in the gap between that for 9 and A.
Lines 630 to 670: are a simple multiplication of the DE register pair contents by 16. All this entails is simply a shift left 4 bits. (Remember that shifting left one bit performs multiplication by two). The sign bit of the product is checked. If set to a '1' then overflow is detected and a branch to 'overflow' is forced.
Line 700: the DE register pair has been shifted left by one nybble so the lower nybble of the E register will be vacant. The accumulator contents, which contains the current hex digit are then placed here by adding the accumulator to the E register.
Lines 730 to 800: are identical to those of 960 to 1040 of the previous program with the exception of line 990. The stack is not used.

Decimal and hexadecimal output routines

In many small machine code programs we tend to rely on PEEK commands from BASIC to display the contents of certain memory locations. Since PEEK can only display the contents of one location, (8 bits) we usually have to resort to quite complex lines of BASIC to display 16 or 32 bit values such as:

PRINT PEEK(&7100)+PEEK(&7101)*256
or
PRINT HEX$(PEEK(&7100)+PEEK(&7101)*256)

The position may of course be further complicated, in the case of decimal output,if the sign of the number is to be taken into account in the final output.

If we wish to have stand alone machine code programs without resorting to BASIC then we need routines that will display the contents of a group of locations in either decimal or hexadecimal digits. The example programs given in this chapter are 16 bit (2 sequential locations), although the principles can easily be extended to display 4 or more sequential bytes in memory.

Decimal output

A worthwhile decimal output routine should convert, say 16 bit, binary numbers to decimal digits with sign detection. The routine should output a '−' minus character prefix if negative. The algorithm used is outlined below:

(a) Check the sign bit of the 16 bit binary number.
(b) If negative, indicated by a '1' in bit 7 of the high byte, then two's complement the number and output a '−' minus sign.
(c) Divide the number by 10 and place the remainder on the stack.

(d) Check if number has reached zero, if not branch back to (c).
(e) Withdraw remainders from stack, convert to ASCII codes by adding 30 hex and output to screen. Note that the most significant digit is placed on the stack last so is extracted first. (Last In First Out, LIFO).

To output, say, the 16 bit number, 32402, the following steps would be occur:

Cycle	*Number*	*Divide by 10*	*Remainder*
1	32402	3240	2
2	3240	324	0
3	324	32	4
4	32	3	2
5	3	0	3

For convenience we have expressed the numbers above in decimal form although within the machine, of course, they are binary. Once the number is reduced to zero the remainders are pulled off the stack, converted to ASCII codes and output.

Program 11.6 is a decimal output routine and can be envisaged as the equivalent of a 16 bit signed PEEK. Before calling the output routine the 16 bit number is assumed transferred to locations &7100 (lowbyte) and &7101 (highbyte). However the HL register pair could be loaded directly with the 16 bit binary number you wish to output.

Program 11.6 Signed decimal output routine (16 bit)

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                     10 ;DECIMAL OUTPUT ROUTINE
                     20 ;(signed 16 bit binary to decimal)
7000                 30 begin:  EQU  #7000
7100                 40 top:    EQU  begin+#100
7100                 50 number: EQU  top
BB5A                 60 OUTPUT: EQU  #BB5A
7000                 70         ORG  begin
7000  2A0071         80         LD   HL,(number)
7003  CB7C           90         BIT  7,H
7005  280C          100         JR   Z,start
7007  3E2D          110         LD   A,"-"
7009  CD5ABB        120         CALL OUTPUT
700C  EB            130         EX   DE,HL
700D  210000        140         LD   HL,0
7010  A7            150         AND  A
7011  ED52          160         SBC  HL,DE
7013  0E0A          170 start:  LD   C,10
7015  1E00          180         LD   E,0
```

```
7017  3E00      190 div10:  LD   A,0
7019  0610      200         LD   B,16
701B  29        210 shift:  ADD  HL,HL
701C  17        220         RLA
701D  B9        230         CP   C
701E  3802      240         JR   C,over
7020  91        250         SUB  C
7021  2C        260         INC  L
7022  10F7      270 over:   DJNZ shift
7024  1C        280         INC  E
7025  F5        290         PUSH AF
7026  7C        300         LD   A,H
7027  B5        310         OR   L
7028  20ED      320         JR   NZ,div10
702A  43        330         LD   B,E
702B  F1        340 loop:   POP  AF
702C  C630      350         ADD  A,#30
702E  CD5ABB    360         CALL OUTPUT
7031  10F8      370         DJNZ loop
7033  C9        380         RET

Pass 2 errors: 00

Table used:    119   from    500
```

Breakdown of Program 11.6

Lines 30 to 70: set the various assembler directives and labelled locations.
Line 80: loads the HL register pair with the 16 bit number to be output.
Lines 90 to 100: check the sign, bit 7 of the high byte of the number, and branch to 'start' if positive (bit 7=0).
Lines 110 to 120: output a '−' minus sign using the firmware routine **TXT OUTPUT.**
Lines 130 to 160: two's complement the 16 bit binary number.
Lines 170 to 280: perform division by 10. The algorithm used has been covered in detail in Chapter 10. Note that register E is used as a byte counter for remainders placed on the stack in line 290.
Line 290: places remainder on the stack after division by 10.
Lines 300 to 320: check if the number has been reduced to zero by repeated integer division by 10. If non zero then a branch is made back to 'div10'.
Line 300: transfers the stack byte counter to the B register so DJNZ can be used in the output loop, lines 340 to 370.
Lines 340 to 370: withdraw the remainders from the stack, convert the values to ASCII codes by adding 30 hex, and output the characters using the firmware routine **TXT OUTPUT.**

Hexadecimal output

A hexadecimal output routine should convert, say, 16 bit binary numbers to hexadecimal digit characters. In many ways hex output

is easier to arrange than decimal. We need not worry about sign, as this is implied. On the other hand the final looped conversion to ASCII characters is a little more complex. The algorithm used is outlined below:

(a) Place the least significant nybble (4 bits) of the number on the stack. To achieve this the high nybble of the byte is first ANDed out (set to zeros).
(b) Divide the number remaining by 16. That is shift right 4 bit positions.
(c) Check if number has reached zero if not branch back to (a).
(d) Output a '&' character signifying what follows are hexadecimal characters.
(e) Withdraw nybbles from the stack convert to ASCII character codes and output to screen. Note again that the most significant digit is placed on the stack last so is extracted first.

As an example, suppose we want to output the 16 bit number &A7D8. The following steps would occur:

Cycle	*Number*	*Divide by 16*	*Stack nybble*
1	A7D8	A7D	8
2	A7D	A7	D
3	A7	A	7
4	A	0	A

For convenience we have expressed the numbers above in hexadecimal form rather than binary. Once the number is reduced to zero the nybbles are pulled off the stack, converted to ASCII codes and the characters output.

Program 11.7 is a hexadecimal output example. Before calling the routine, the 16 bit number is assumed transferred to locations &7100 (lowbyte) and &7101 (highbyte). However, if preferred, the HL register pair could be loaded directly with the 16 bit binary number you wish to output.

Program 11.7 Hexadecimal output routine (16 bit)

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                       10 ;HEXADECIMAL OUTPUT ROUTINE
                       20 ;(16 bit binary to hex characters)
7000                   30 begin:   EQU  #7000
7100                   40 top:     EQU  begin+#100
```

```
7100                 50 number: EQU  top
BB5A                 60 OUTPUT: EQU  #BB5A
7000                 70         ORG  begin
7000  0E00           80         LD   C,0
7002  2A0071         90         LD   HL,(number)
7005  7D            100 loop:   LD   A,L
7006  E60F          110         AND  #0F
7008  F5            120         PUSH AF
7009  0C            130         INC  C
700A  0604          140         LD   B,4
700C  CB3C          150 div16:  SRL  H
700E  CB1D          160         RR   L
7010  10FA          170         DJNZ div16
7012  7C            180         LD   A,H
7013  B5            190         OR   L
7014  20EF          200         JR   NZ,loop
7016  3E26          210         LD   A,"&"
7018  CD5ABB        220         CALL OUTPUT
701B  41            230         LD   B,C
701C  F1            240 loop2:  POP  AF
701D  FE0A          250         CP   10
701F  3802          260         JR   C,over
7021  C607          270         ADD  A,7
7023  C630          280 over:   ADD  A,#30
7025  CD5ABB        290         CALL OUTPUT
7028  10F2          300         DJNZ loop2
702A  C9            310         RET

Pass 2 errors: 00

Table used:   107  from   500
```

Breakdown of Program 11.7

Lines 30 to 70: set the various assembler directives and labelled locations.
Line 80: initialise the hex digit counter to zero.
Line 90: loads the HL register pair with the 16 bit number to be output.
Line 100: copies the L register contents into the accumulator.
Lines 110 to 130: ANDS out the high nybble of the accumulator contents and places the low nybble, left in the accumulator, on the stack. The digit counter is then incremented.
Lines 140 to 170: perform a simple division by 16 by shifting the number in the HL register pair 4 bit positions to the right.
Lines 180 to 200: check if the number has been reduced to zero by repeated integer division by 16. If non zero then a branch is made back to 'div16'.
Lines 210 to 220: output a '&' character.
Line 230: transfers the hex digit counter to the B register, so DJNZ can be used in the output loop.

Line 240: designates the start of the output loop and each cycle, withdraws a nybble from the stack into the accumulator.
Lines 250 to 260: check if the accumulator contents is less than 10. If so a branch is made to 'over'.
Line 270: if this instruction is executed then the accumulator contents are greater than or equal to 10. That is to say ABCDE or F. Thus 7 is added to the accumulator contents to allow for the gap between the ASCII codes of 9 and A.
Line 280: adds 30 hex to the accumulator contents to convert the digit to its ASCII character code.
Line 290: outputs the character using the firmware routine **TXT OUTPUT.**
Line 300: loops back for the next stack withdrawal if the digit counter is non zero.

Summary

1 A string input routine should display a prompt, echo the character to the screen, allow back deletion and warn when limits are exceeded.
2 A decimal input routine should cater for leading '+' or '–' characters, detect overflow and reject invalid characters.
3 A decimal output routine should handle, at least, 16 bit numbers, and display sign as well as magnitude.
4 The sign of a hexadecimal number is implied by the number itself so output routines do not require a sign character.
5 If the first hex digit is greater than 7, the number is negative.

12 Parameter passing and introduction to resident system extensions

The passing of parameters to and from machine code subroutines often leads to confusion. In assembly language we have a choice of methods but when calling machine code subroutines from BASIC we are restricted to the versatile, CALL <address> <parameter list>, command. As an example of exploiting this command, a fast machine code sort routine is provided which is capable of sorting a BASIC string array of a thousand elements in just over a second. The chapter continues with an introduction to Resident System Extensions although we will leave the more complex applications to Chapter 13.

Passing parameters in assembly language

In assembly language, we have seen that a subroutine is invoked via the use of a **CALL <address>** instruction. It may be required that data be passed to a subroutine in order for it to perforn its function. Moreover, we may wish to pass resultant data back to the calling program. In this sense a 'data' element is referred to as a parameter and a group of 'data' elements as a parameter list. When calling subroutines it is necessary to:

1 Set up any parameters that are to be accessed by the subroutine before using the **CALL** instruction.
2 Set up parameters (if any) that are to be accessed by the calling program on return.

We shall describe four methods of passing parameters to/from subroutines in assembly language. They are via:

(a) Registers
The Amstrad operating system, in common with most other firmware, uses this method of passing parameters to/from subroutines.

If an 8 bit value is to be passed, then the accumulator is the natural choice. For 16 bit values a register pair such as DE could be

employed. The subroutine will be written on the assumption that its parameters will be available in certain registers when called. The calling program equally expects returned parameters (if any) to be available in certain registers.

This method suffers from the disadvantage that the registers used may already hold important information. The solution, of course,is to save the contents of such registers on the stack before calling and restore them on return. On entry to the subroutine it is good practice to save on the stack any registers, excluding those used to pass parameters, that are used by the subroutine itself. The registers can then be restored before return.

(b) Parameter blocks

The Amstrad **CALL** command uses this method of passing parameters from BASIC to machine code subroutines. Where an assortment of data needs to be passed to a subroutine, a parameter block (data list) is used. The base address of the block of consecutive addresses is normally passed via one of the index registers, IX or IY. For example, if the base address of the parameter block is #7100 then the IX register can be set to #7100 by the instruction **LD IX,#7100**. Individual parameters can then be picked up by the subroutine, as required, by setting the displacement constant (d) in the **LD reg,(IX+d)** instruction. For example, if IX contained #7100 as above, then **LD A,(IX+4)** would load the accumulator with the contents of address #7104. Any location offset from the parameter block's base address can thus be accessed. However, it should be remembered that there is a limit of 255 for the displacement byte so it is important that notes be kept on the relative positions of parameters within the block.

(c) Fixed locations

This method, due to its inherent simplicity, is probably the most popular method of passing parameters to and from subroutines. The subroutine is written to expect its parameters to be left in certain fixed locations. The calling program will also expect returned parameters to be in fixed locations. There is no need to save or restore registers in the calling program but it is still a good idea to save, on the stack, registers used in the subroutine on entry. They can be restored immediately before return. In some circles, this practice has been known to cause the odd monocle to drop to the end of its string. Academics say, with some truth, that it is difficult to keep subroutine libraries on disc or tape and there may be frequent clashes between chosen locations and labels. Nevertheless the practice is still widely used. It is usually a trivial task to tailor a subroutine to suit a specific program because the inclusion of 'search and replace' facilities are now quite common in assemblers.

(d) The stack

It is also possible to pass parameters via the stack. However, some

frightful muddles may occur if we forget that, before entry to the subroutine, the return address is placed on the stack. Therefore we must make sure, before accessing parameters, that the return address is first popped from the stack and stored within the subroutine. Finally, after returned parameters have been pushed on the stack, the return address must be replaced. On encountering the RET instruction the program counter will be loaded from the top of the stack which will then be the return address.

Executing machine code subroutines from BASIC.

When designing subroutines intended to be called from BASIC, the **CALL** command is brought into service. The format is

CALL <address>,<parameter list>
For example:
CALL &7000,a%,b%,c%

If, in your programs, a parameter is to be returned to BASIC then the PEEK command may be used.

Instead of treating the **CALL** command in a general sense, a practical example, embodying most eventualities, will be more helpful. Although this may seem a double barrelled approach, the part that is of no immediate interest can be temporarily ignored during the first reading. The example chosen is a fast sorting routine and details of how to use and pass parameters will be given in the text.

Machine code sorting of BASIC string arrays

When programming in BASIC, it is often required to sort a series of strings held in array form. However, sorting is a time consuming process, and the execution time can be unacceptably high with large arrays. We are, after all, up against the inherent defects of the BASIC language. It is an interpretive, rather than a compiled, language so execution speed is likely to be slow. Even in machine code, it will still be important to select a suitable algorithm. For example, assuming a list is large, the common bubble sort written in machine code may not execute all that much faster than a more efficient algorithm written in BASIC.

Choosing a sorting algorithm

One of the fastest known methods of sorting large contiguous lists (arrays) is by means of an algorithm called the 'Quicksort', devised by C.A.R. Hoare in 1962. A machine code version of the Quicksort that can be called from any BASIC program will provide a useful addition to any subroutine library.

The Quicksort algorithm

The Quicksort is rather complicated so a brief overview of the main features are first given in plain English.

The fundamental idea behind the Quicksort is that the sorting of *small lists* is more efficient (fewer comparisons) than sorting large lists. It follows, that if we split an array into two *sublists,* containing different groups of strings, and sort each list separately, we reduce the number of comparisons and therefore the sorting time. To do this, we estimate an array element that, hopefully, will have a median value and call this the *pivot*. All strings having a value less than the pivot (ASCII wise) will be placed above it in the lower half of the array and all strings having a greater value than the pivot placed in the higher half. If these two portions of the array are sorted separately on *either side of the pivot* then the array will be completely sorted.

The estimate of the pivot value is all important. For instance, we could choose the first array element or the last array element in a random array. However,if the list is partially sorted, as may occur in practice, the performance may be seriously degraded because the pivot will be too far offset from the median value to make the split worthwhile. This possibility can be reduced (statistically) by choosing the pivot as the mid point element of the array. We should mention that there are other ways of obtaining the pivot but we will employ this one. Incidentally, during worst case conditions, where the above effect is dominant, the sorting time can be as poor as that for a bubble sort, (approximately proportional to n^2, where n= number of elements) but this is unlikely to happen in practice.

One method of implementing Quicksort is to keep partitioning lists in this way till we have many lists containing, say, 15 elements at most and then bubble sorting them. Alternatively, if we carry on and take this partioning process to its limit then each sublist will eventually contain only one element. In this case there will be no need to employ a bubble sort at all since a list containing only one element must already be sorted!

The practical implementation of Quicksort relies heavily on the use of the stack for storing the limits of lists not yet sorted. (Remember that the stack is a section of memory where the last variable value stored is the first recalled). To simplify the description, we will use the following labels:

head	= first array element to be included in sort.
tail	= last array element to be included in sort.
lowhead	= first array element in lower sublist.
lowtail	= last array element in lower sublist.
highhead	= first array element in higher sublist.
hightail	= last array element in higher sublist.

The head and tail parameters of the sublists yet to be sorted are put on the stack. Thus, if we are presently sorting the array between

A$(lowhead) and A$(lowtail) then highhead and hightail would be placed on the stack so that the sublist, A$(highhead) to A$(hightail), could be sorted later. It transpires that it is better to put the *longer* sublist limits on the stack and process the *shorter* sublist immediately. Each time a list is further partitioned within the loop, the process is repeated. On exit from the loop the sublists, whose parameters were placed on the stack, are taken in sequence (last in, first out) and sorted in a similar manner. In view of the complexity of the algorithm, and the danger of lapsing into verbal excesses, a more detailed flowchart, Figure 12.1 (overleaf), is provided to help in tracing the assembly code listing Program 12.1.

Although adequate in performance, the program was written with clarity, rather than efficiency, in mind. When developing large programs, it is relatively easy to save a few clock cycles or a byte here and there if you have plenty of time to spare although the law of diminishing returns will eventually take over.

Fig 12.1 Flowchart of Program 12.1

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                         10 ;QUICKSORT
                         20 ;OF A STRING ARRAY
7000                     30 begin:   EQU   #7000
7203                     40 top:     EQU   begin+#0203
7203                     50 tail:    EQU   top
7205                     60 head:    EQU   top+2
7207                     70 array:   EQU   top+4
7209                     80 stackc:  EQU   top+6
720A                     90 plen:    EQU   top+7
720B                    100 ppoint:  EQU   top+8
720D                    110 sd1poi:  EQU   top+10
720F                    120 sd2poi:  EQU   top+12
7000                    130          ORG   begin
                        140 ;Pick up and store base address
                        150 ;of string descriptors : array
7000  DD6E00            160 sort:    LD    L,(IX)
7003  DD6601            170          LD    H,(IX+1)
7006  220772            180          LD    (array),HL
                        190 ;Pick up tail of array : tail
7009  DD6E02            200          LD    L,(IX+2)
700C  DD6603            210          LD    H,(IX+3)
700F  220372            220          LD    (tail),HL
                        230 ;Pick up head of array : head
7012  DD6E04            240          LD    L,(IX+4)
7015  DD6605            250          LD    H,(IX+5)
7018  220572            260          LD    (head),HL
                        270 ;Set stackcounter to zero
701B  97                280          SUB   A
701C  320972            290          LD    (stackcount),A
                        300 ;Branch to bypass if head>=tail
```

```
701F  2A0572     310 loop:   LD   HL,(head)
7022  ED5B0372   320         LD   DE,(tail)
7026  A7         330         AND  A
7027  ED52       340         SBC  HL,DE
7029  D23D71     350         JP   NC,bypass
                 360 ;Initialise highhead and lowtail
                 370 ; IX & IY respectively
702C  DD2A0572   380         LD   IX,(head)
7030  FD2A0372   390         LD   IY,(tail)
                 400 ;Calculate pivot string
                 410 ;descriptor address : HL
7034  2A0572     420         LD   HL,(head)
7037  ED5B0372   430         LD   DE,(tail)
703B  19         440         ADD  HL,DE
703C  CB3C       450         SRL  H
703E  CB1D       460         RR   L
7040  54         470         LD   D,H
7041  5D         480         LD   E,L
7042  29         490         ADD  HL,HL
7043  19         500         ADD  HL,DE
7044  ED4B0772   510         LD   BC,(array)
7048  09         520         ADD  HL,BC
                 530 ;Get length and address of
                 540 ;pivot string : plen & ppoint
7049  7E         550         LD   A,(HL)
704A  320A72     560         LD   (plen),A



Hisoft GENA3.1 Assembler. Page     2.


704D  23         570         INC  HL
704E  5E         580         LD   E,(HL)
704F  23         590         INC  HL
7050  56         600         LD   D,(HL)
7051  ED530B72   610         LD   (ppoint),DE
                 620 ;Set pointer to first string
                 630 ;descriptor: sd1point
7055  2A0572     640         LD   HL,(head)
7058  54         650         LD   D,H
7059  5D         660         LD   E,L
705A  29         670         ADD  HL,HL
705B  19         680         ADD  HL,DE
705C  09         690         ADD  HL,BC
705D  220D72     700         LD   (sd1point),HL
                 710 ;Set pointer to second string
                 720 ;descriptor: sd2point
7060  2A0372     730         LD   HL,(tail)
7063  54         740         LD   D,H
7064  5D         750         LD   E,L
7065  29         760         ADD  HL,HL
7066  19         770         ADD  HL,DE
7067  09         780         ADD  HL,BC
7068  220F72     790         LD   (sd2point),HL
                 800 ;Get length and address of first
                 810 ;string: C & DE
```

```
706B  2A0D72     820 first:  LD   HL,(sd1point)
706E  4E         830         LD   C,(HL)
706F  23         840         INC  HL
7070  5E         850         LD   E,(HL)
7071  23         860         INC  HL
7072  56         870         LD   D,(HL)
                 880 ;Compare first string to pivot
                 890 ;branch to proc1 if string<pivot
7073  0600       900         LD   B,0
7075  2A0B72     910         LD   HL,(ppoint)
7078  1A         920 comp:   LD   A,(DE)
7079  BE         930         CP   (HL)
707A  3811       940         JR   C,proc1
707C  201C       950         JR   NZ,second
707E  04         960         INC  B
707F  3A0A72     970         LD   A,(plen)
7082  B8         980         CP   B
7083  2815       990         JR   Z,second
7085  79        1000         LD   A,C
7086  B8        1010         CP   B
7087  2804      1020         JR   Z,proc1
7089  13        1030         INC  DE
708A  23        1040         INC  HL
708B  20EB      1050         JR   NZ,comp
                1060 ;Add 3 to sd1point
708D  2A0D72    1070 proc1:  LD   HL,(sd1point)
7090  23        1080         INC  HL
7091  23        1090         INC  HL
7092  23        1100         INC  HL
7093  220D72    1110         LD   (sd1point),HL
                1120 ;increment highhead
7096  DD23      1130         INC  IX
7098  18D1      1140         JR   first


Hisoft GENA3.1 Assembler. Page     3.


                1150 ;Get length and address of
                1160 ;second string : C & DE
709A  2A0F72    1170 second: LD   HL,(sd2point)
709D  4E        1180         LD   C,(HL)
709E  23        1190         INC  HL
709F  5E        1200         LD   E,(HL)
70A0  23        1210         INC  HL
70A1  56        1220         LD   D,(HL)
                1230 ;Compare second string to pivot
                1240 ;branch to proc2 if string>pivot
70A2  0600      1250         LD   B,0
70A4  EB        1260         EX   DE,HL
70A5  ED5B0B72  1270         LD   DE,(ppoint)
70A9  1A        1280 comp2:  LD   A,(DE)
70AA  BE        1290         CP   (HL)
70AB  3811      1300         JR   C,proc2
70AD  201C      1310         JR   NZ,over
70AF  04        1320         INC  B
70B0  79        1330         LD   A,C
```

```
70B1  B8        1340          CP   B
70B2  2817      1350          JR   Z,over
70B4  3A0A72    1360          LD   A,(plen)
70B7  B8        1370          CP   B
70B8  2804      1380          JR   Z,proc2
70BA  13        1390          INC  DE
70BB  23        1400          INC  HL
70BC  20EB      1410          JR   NZ,comp2
                1420 ;Subtract 3 from sd2point
70BE  2A0F72    1430 proc2:   LD   HL,(sd2point)
70C1  2B        1440          DEC  HL
70C2  2B        1450          DEC  HL
70C3  2B        1460          DEC  HL
70C4  220F72    1470          LD   (sd2point),HL
                1480 ;decrement lowtail
70C7  FD2B      1490          DEC  IY
70C9  18CF      1500          JR   second
                1510 ;Compare sd1point to sd2point
                1520 ;Br. proc3 if sd1point<sd2point
                1530 ;Br. skip if sd1point>sd2point
                1540 ;if = dec lowtail & inc highhead
70CB  A7        1550 over:    AND  A
70CC  2A0D72    1560          LD   HL,(sd1point)
70CF  ED5B0F72  1570          LD   DE,(sd2point)
70D3  ED52      1580          SBC  HL,DE
70D5  3808      1590          JR   C,proc3
70D7  202D      1600          JR   NZ,skip
70D9  FD2B      1610          DEC  IY
70DB  DD23      1620          INC  IX
70DD  1827      1630          JR   skip
                1640 ;swop string descriptors
70DF  0603      1650 proc3:   LD   B,3
70E1  2A0D72    1660          LD   HL,(sd1point)
70E4  1A        1670 swop:    LD   A,(DE)
70E5  4E        1680          LD   C,(HL)
70E6  EB        1690          EX   DE,HL
70E7  71        1700          LD   (HL),C
70E8  12        1710          LD   (DE),A
70E9  13        1720          INC  DE


Hisoft GENA3.1 Assembler. Page     4.

70EA  23        1730          INC  HL
70EB  10F7      1740          DJNZ swop
                1750 ;Add 3 to sd1point
70ED  2A0D72    1760          LD   HL,(sd1point)
70F0  23        1770          INC  HL
70F1  23        1780          INC  HL
70F2  23        1790          INC  HL
70F3  220D72    1800          LD   (sd1point),HL
                1810 ;subtract 3 from sd2point
70F6  2A0F72    1820          LD   HL,(sd2point)
70F9  2B        1830          DEC  HL
70FA  2B        1840          DEC  HL
70FB  2B        1850          DEC  HL
```

```
70FC  220F72    1860          LD   (sd2point),HL
                1870 ;dec highhead:inc lowtail
70FF  DD23      1880          INC  IX
7101  FD2B      1890          DEC  IY
7103  C36B70    1900          JP   first
                1910 ;increment stackcounter
7106  210972    1920 skip:    LD   HL,stackcount
7109  34        1930          INC  (HL)
                1940 ;Calc lowtail-lowhead &
                1950 ; hightail-highead
710A  A7        1960          AND  A
710B  FDE5      1970          PUSH IY
710D  E1        1980          POP  HL
710E  ED5B0572  1990          LD   DE,(head)
7112  ED52      2000          SBC  HL,DE
7114  EB        2010          EX   DE,HL
7115  A7        2020          AND  A
7116  2A0372    2030          LD   HL,(tail)
7119  DDE5      2040          PUSH IX
711B  C1        2050          POP  BC
711C  ED42      2060          SBC  HL,BC
                2070 ;Compare results & stack larger
                2080 ;limits : process smaller limits
711E  A7        2090          AND  A
711F  ED52      2100          SBC  HL,DE
7121  300D      2110          JR   NC,sthigh
7123  2A0572    2120          LD   HL,(head)
7126  E5        2130          PUSH HL
7127  FDE5      2140          PUSH IY
7129  DD220572  2150          LD   (head),IX
712D  C31F70    2160          JP   loop
7130  DDE5      2170 sthigh:  PUSH IX
7132  2A0372    2180          LD   HL,(tail)
7135  E5        2190          PUSH HL
7136  FD220372  2200          LD   (tail),IY
713A  C31F70    2210          JP   loop
                2220 ;compare stackcounter to zero
713D  3A0972    2230 bypass:  LD   A,(stackcount)
7140  FE00      2240          CP   0
7142  280F      2250          JR   Z,finish
                2260 ;decrement stackcounter
7144  3D        2270          DEC  A
7145  320972    2280          LD   (stackcount),A
                2290 ;Pop tail & head from stack
7148  E1        2300          POP  HL

Hisoft GENA3.1 Assembler. Page     5.

7149  220372    2310          LD   (tail),HL
714C  E1        2320          POP  HL
714D  220572    2330          LD   (head),HL
7150  C31F70    2340          JP   loop
7153  C9        2350 finish:  RET

Pass 2 errors: 00

Table used:   310  from   500
```

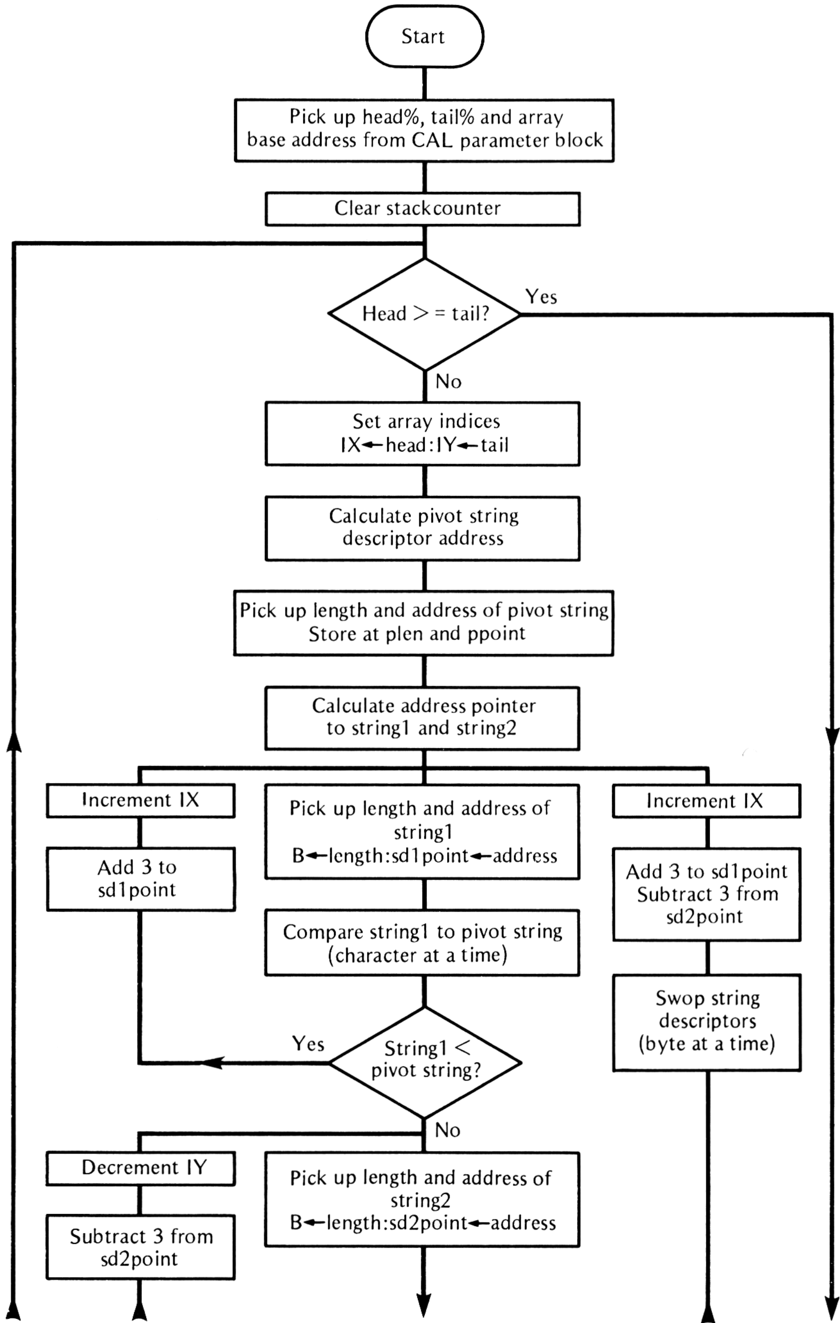
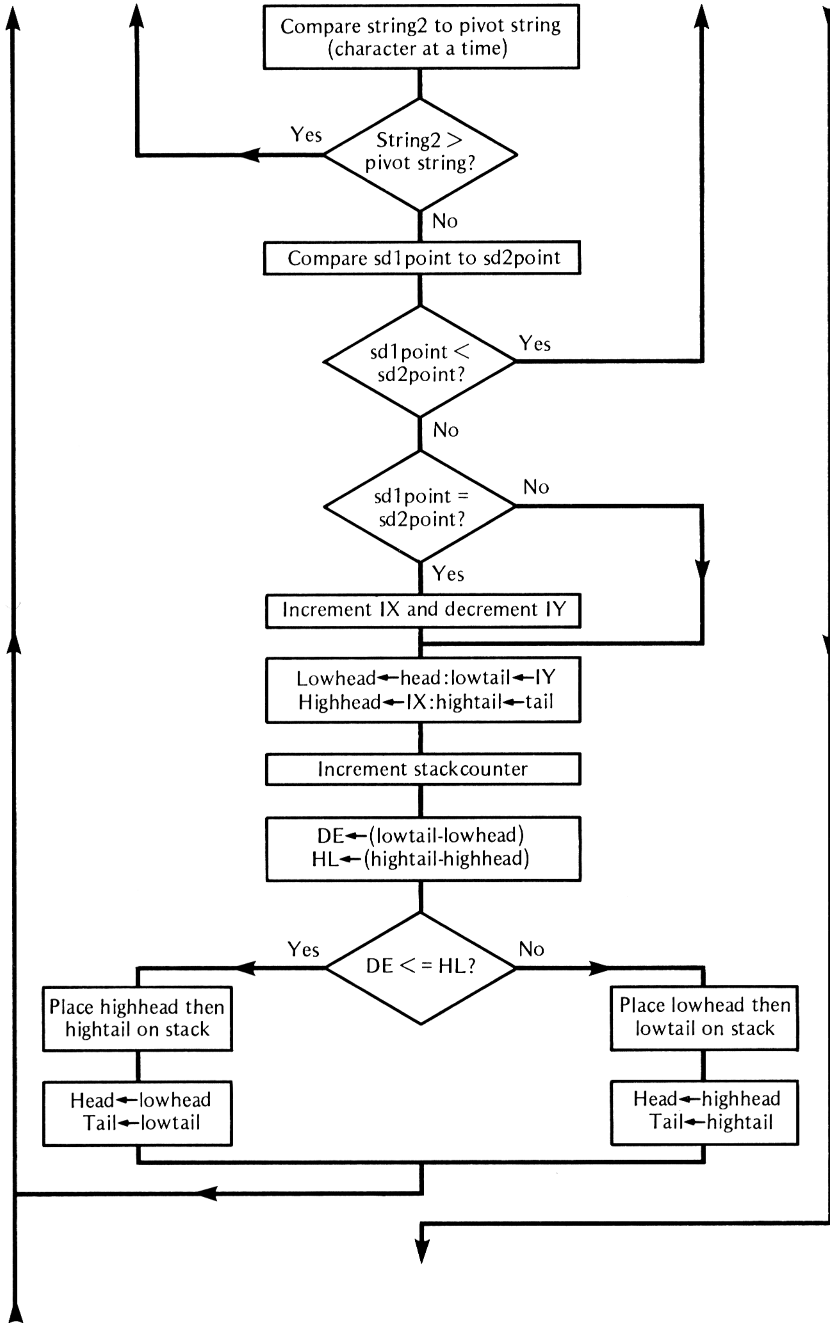


Fig. 12.1 Flowchart of Program 12.1


Compare string2 to pivot string (character at a time)
String2 > pivot string?
Yes
No
Compare sd1point to sd2point
sd1point < sd2point?
Yes
No
sd1point = sd2point?
No
Yes
Increment IX and decrement IY
Lowhead←head:lowtail←IY
Highhead←IX:hightail←tail
Increment stackcounter
DE←(lowtail-lowhead)
HL←(hightail-highhead)
DE < = HL?
Yes
No
Place highhead then hightail on stack
Head←lowhead
Tail←lowtail
Place lowhead then lowtail on stack
Head←highhead
Tail←hightail

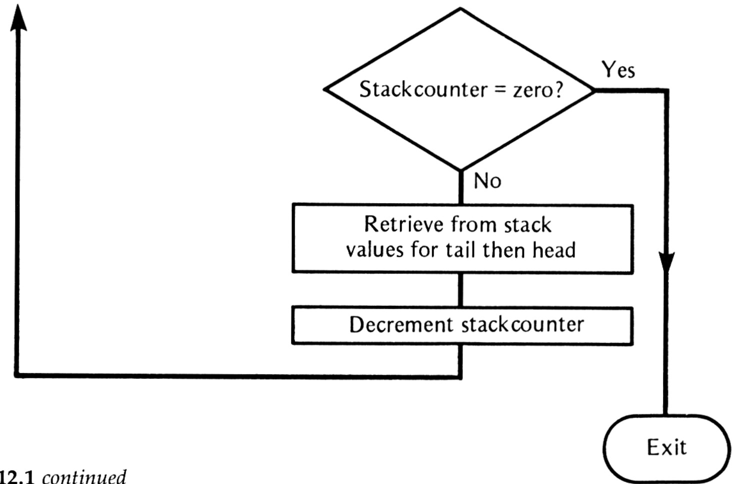


Fig. 12.1 *continued*

Machine code string array sort

When a string array is DIMensioned by the BASIC interpreter, an access table is set up containing three bytes for each string element. These bytes are not the strings themselves but the length and address of where they are stored. They are called the String Descriptors and represent the string length, low byte address and high byte address respectively. The string length byte is the lowest memory. Figure 12.2 shows how an array is set up in memory by the BASIC interpreter.

The strings themselves, in the form of ASCII characters, are stored immediately below HIMEM (which Amstrad refers to as the 'heap') in sequential memory locations, the address of which is given in bytes 2 and 3 of the string descriptor. Since the string descriptors are at a fixed distance apart (3 bytes), any particular string (x) can be accessed by multiplying the base address of the access table by 3x. A string array is thus formed by a series of such string descriptors stored sequentially in memory. Swapping strings during a string sort is not as difficult as it may first appear. We can leave the strings themselves where they are in the heap and swap the *string descriptors*, since these tell the BASIC interpreter where the actual strings are stored.

Program 12.1 is the assembly code listing. The object code is assembled at address &7000 onwards. A suitable test is provided by Program 12.2. It loads the machine code binary file from tape or disc, sets up a random string array, calls the machine code routine and displays the sorted array.

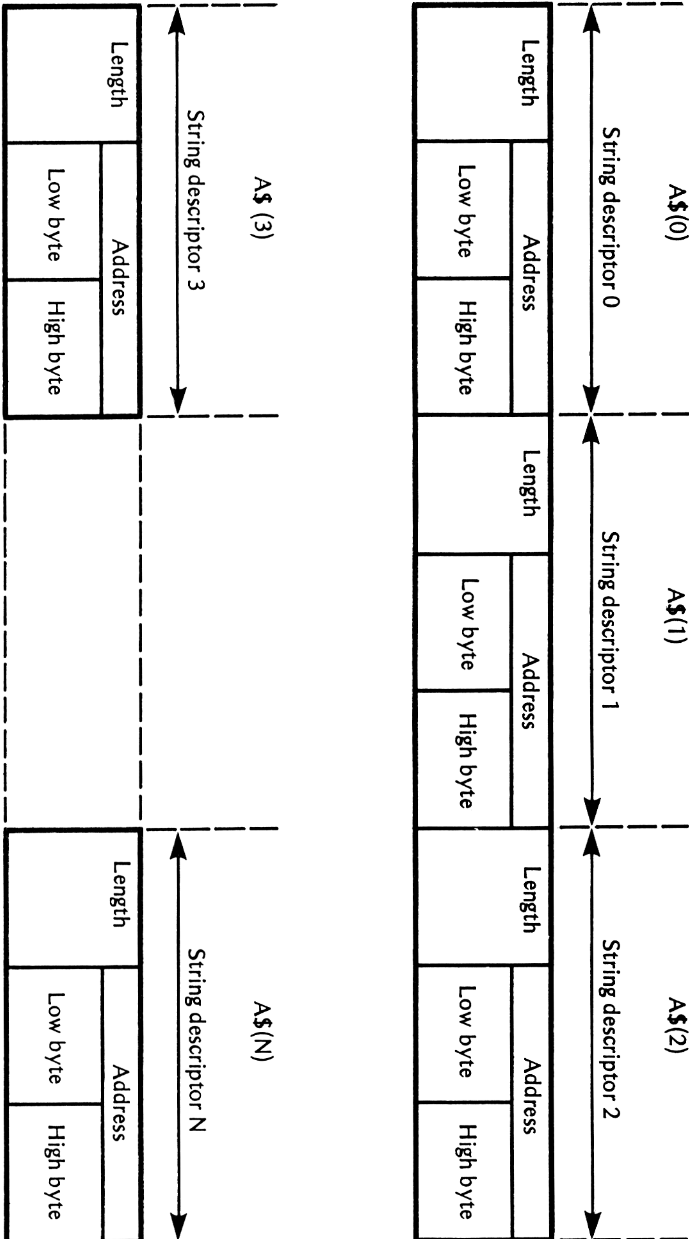


Fig. 12.2 String descriptors

Program 12.2 Test program for Quicksort

```
10 REM TEST PROGRAM: STRING SORT
20 CLS
30 MEMORY &6FFF
40 LOAD "SORT.BIN",&7000
50 CLS
60 INPUT"Sort how many strings";NUMBER%
70 PRINT
80 REM FILL AND DISPLAY RANDOM ARRAY
90 DIM A$(NUMBER%)
100 head%=1:tail%=NUMBER%
110 FOR R%=head% TO tail%
120 B$=""
130 A%=6*RND+1
140 FOR Z%=1 TO A%
150 N%=25*RND
160 K$=CHR$(N%+65)
170 B$=B$+K$
180 NEXT
190 A$(R%)=B$
200 PRINT A$(R%)
210 NEXT
220 PRINT:PRINT
230 PRINT"SORTING ARRAY"
240 PRINT:PRINT
250 START=TIME/300
260 CALL &7000,head%,tail%,@A$(0)
270 T=TIME/300-START
280 FOR R%=head% TO tail%
290 PRINT A$(R%)
300 NEXT
310 PRINT
320 PRINT"STRINGS SORTED=";NUMBER%
330 PRINT
340 PRINT"SORTING TIME=";ROUND(T,2);"SECONDS"
350 PRINT
360 INPUT"Another test (Y/N)";K$
370 K$=UPPER$(K$)
380 IF K$="Y" THEN ERASE A$:GOTO 50 ELSE END
```

Producing a binary file of the object code

The most convenient way to use machine code subroutines from within a BASIC program is to produce a binary file of the object code on tape or disc which can then be loaded automatically by the first few lines of the BASIC program. This file can be produced by the following method:

(a) Load the HiSoft DEVPAC asembler at, say, address &2000 hex.
(b) Type in the assembly code listing, Program 12.1.
(c) Save a copy of the source code on tape (or disc) by typing:

P 10,2350,SORT.SRC

Note that the filenames are not enveloped in double quotes as is normal with BASIC programs.

The source code file can be reloaded at some later time with:

G,,SORT.SRC

(d) Assemble the source code by typing the assembler command, **A**. When asked for table size, respond with 500, which allows more than adequate space for the symbol table.
(e) Clear typing errors if any errors are reported.
(f) Produce an object code file that can be loaded directly from tape (or disc) by a main program. This can be done automatically by typing the assembler command:

O,,SORT.BIN

(g) Perform a hard reset to clear memory.
(h) Type in and save on disc the test program, Program 12.2. This will automatically load, run and test the binary file you have just produced.

The object code itself is loaded into a section of memory reserved above HIMEM at &7000. This allows sufficient memory for the object code and, in the case of the CPC464, a permanently allocated cassette buffer area.

Relocation of machine code

The main disadvantage of Program 12.1 as it stands is that the object code will only execute at address &7000. Loading it elsewhere in memory will result in chaos because the object code is not relocatable. If, for any reason you need the code located in memory, other than &7000, you must reassemble at an alternative address. This is an easy task with Program 12.1 – simply change the address in the **EQU** statement in line 30 and reassemble. Chapter 13 will show how to alter the program so that it will produce self relocating object code.

Calling the routine from BASIC

To **CALL** the routine it is necessary to pass over a few parameters. This is accomplished by the insertion of one line in your BASIC program. For instance.

CALL &7000,head%,tail%,@A$(0)

will sort the array A$ between the limits head% and tail%. If you

have more than one array in memory then either or both may be sorted with the same routine. For example:

CALL &7000,head%,tail%,@B$(0)

will force a sort of the array B$ without any modifications to the subroutine itself. This of course, would not be possible in BASIC, because a separate subroutine would be needed for each array. The &7000 term is the execution address of the subroutine. This can be altered if you assemble the code elsewhere in memory. The integer variable, head%, is the array index to the first item in the array (Usually set to 1).

The integer variable, tail%, is the highest array index. That is to say, if head% is set to 1 and tail% is set to 200 then the above call will sort the array A$ from A$(1) to A$(200). Even parts of arrays may be sorted if head% and tail% are set to index subsections of the array.

The @A$(0) parameter passes over the address of the element A$(0) which is the base address of the array. Note that the '@' symbol preceding a variable, means pass over the address of the variable to the machine code routine, not the variable value itself. It is also important to remember that the address of the string descriptor is returned, not the address of the string itself.

When a **CALL** statement is executed in BASIC, the parameter list values following the execution address, are passed over to the machine code subroutine. These are automatically stored in a parameter block, offset from an address stored in the IX register of the Z80 microprocessor. The parameters are offset from the IX register contents in the reverse order to that which they appear in the CALL statement. For instance, with the above CALL, the address of the A$(0) string descriptor (2 bytes) is set up in (IX) and (IX+1), the actual 16 bit value of tail% is set up in (1X+2) and (IX+3) and finally the 16 bit value of head% is set up in (IX+4) and (IX+5). The order being low byte first, high byte second. These parameters may then be accessed, where they are as needed, or picked up and stored in a more convenient section of memory by the subroutine itself.

To give an idea of the execution speed that can be expected from Program, 12.1 see the following table.

Typical execution times for Quicksort	
String array size	*Sorting time(sec)*
100	0.13
200	0.26
300	0.42
500	0.71
1000	1.53
2000	3.38
3000	5.62

Notice the approximately linear relationship between execution time and the number of strings sorted. For comparison, the equivalent algorithm was programmed in BASIC and it took 54 seconds to sort 500 and 230 seconds to sort a thousand strings.

Breakdown of program 12.1

The main labels, referred to in the breakdown are as specified above regarding head, tail, lowhead, lowtail, highhead and hightail. Some of these labels are not used in the listing itself but are present in remarks. We hope this will aid, in conjunction with Figure 12.1, the understanding of the program flow.

To simplify the description, the following definitions apply:

string1: term given to the current string accessed in the list/sublist when scanning the list from the top towards the pivot string position.
string2: term given to the current string accessed in the list/sublist when scanning from the bottom towards the pivot string position.
pivot string: term given to the string at the mid position of the current list/sublist to which string1 and string2 is repeatedly compared.

Line 30: sets the assembly address at #7000.
Lines 50 to 120: assign labels to often used locations. To elaborate on the labels used see the following list:

top: start of data storage area.
array: base address of string descriptors.
Stack: stackcounter for the number of sublist limits placed on the stack, corresponding to sublists yet to be sorted.
plen: length of the pivot string in bytes.
ppoint: address pointer to the current pivot string.
sd1point: address pointer to the array string descriptor, scanning from the head of the current list/sublist.
sd2point: address pointer to the array string descriptor, scanning from the tail of the current list/sublist.

Lines 160 to 180: pick up the base address of the array from the CALL parameter block and store it, more conveniently, in the two sequential locations labelled 'array'.
Lines 200 to 260: pick up the 16 bit values of the BASIC variables tail% and head% from the parameter block and store them at the locations labelled 'tail' and 'head' respectively.
Lines 280 to 290: initialise the stackcounter to zero.
Lines 310 to 350: compare the contents of 'head' and 'tail'. If 'head' is greater than or equal to 'tail' a forward branch to 'bypass' is made.
Lines 380 to 390: initialise the IX and IY registers to the current values, stored at 'head' and 'tail' respectively. On exit from the loop, or scan of the list/sublist, the terminal contents of IX and IY will be the previously defined values, 'highhead' and 'lowtail',

respectively. This saves using separate labelled locations for these values as we have implied in the flowchart.
Lines 420 to 520: calculate the address of the pivot string descriptor, firstly by adding 'head' and 'tail' together. Integer division by 2 is then performed by a single shift right. Finally, this result is multiplied by 3, the number of bytes in a string descriptor, and added to the access table base address 'array'. The pivot string descriptor address is available in the HL register pair. The formula evaluated is:

pivot descriptor address=[(head+tail)/2]×3

where the integer value is taken within the square brackets.
Lines 550 to 610: use implied addressing to pick up the length and address of the pivot string from the string descriptor. The length is stored at 'plen' and the 16 bit address is stored at 'ppoint'.
Lines 640 to 700: calculate the address pointer to the current string1 descriptor and store the result at 'sd1point'. The equation evaluated is:

sd1point=(head×3)+array

Lines 730 to 790: calculate the address pointer to the current string2 descriptor and store the result at 'sd2point'. The equation evaluated is:

sd2point=(tail×3)+array

Lines 820 to 870: use implied addressing to pick up the length and address of string1. The length is placed in register C and the address is in DE.
Lines 900 to 1050: use implied addressing to compare, character by character, string1 to the pivot string. If string1 is less than the pivot string then a branch to 'proc1' is made. This signifies that the current string1 is already on the correct side of the pivot. The loop terminates when a string greater than or equal to the pivot string is found. The string comparisons are made using the following rules:

(a) Two strings are equal when they are the same length and contain the same characters.
(b) The first string is less than the second if they are equal up to the first string but the second string is longer.
(c) One string is less than the other when the first character to differ in one string has a lower ASCII code than that of the other.

Lines 1070 to 1110: add three to the address pointer 'sd1point' so as to point to the next higher string descriptor of the current string1.
Lines 1130 to 1140: increment the array index to string1 in register IX. This is followed by an unconditional jump to 'first' so the next indexed string, scanning down from the top of the current list, can be compared to the pivot string.

Lines 1170 to 1220: use implied addressing to pick up the length and address of string2. The length is placed in register C and the address is in DE.
Lines 1250 to 1410: use implied addressing to compare, character by character, string2 to the pivot string. If string2 is greater than the pivot string then a branch to 'proc2' is made. This signifies that the current string2 is already on the correct side of the pivot. The loop terminates when a string less than or equal to the pivot string is encountered.
Lines 1430 to 1470: subtract three from the address pointer 'sd2point' so as to point to the next lower string descriptor of the current string2.
Lines 1490 to 1500: decrement the array index to string2 in the IY register. This is followed by an unconditional jump to 'second' so the next indexed string, scanning up from the bottom of the current list, can be compared to the pivot string.
Lines 1550 to 1630: compare 'sd1point' with 'sd2point' in various ways. On exit from the above loops, a decision has to be made whether to swap them over or not. If 'sd1point' is less than 'sd2point' then string1 and string2 are swapped to the correct side of the pivot in the list/sublist by a branch to 'proc3'.

As shown in the flowchart, this outer loop, and thus the scans, terminate only if 'sd1point' is greater than or equal to 'sd2point'. If 'sd1point' is equal to 'sd2point' (scans meet) it signifies that the associated string in that array position is equal to the pivot string and is excluded from either sublist. Consequently, the array index, IX, is incremented, IY decremented and a branch to 'skip' made. If 'sd1point' is greater than 'sd2point' at this point then the scans have crossed. Any strings corresponding to the crossover section, excluding those currently indexed by IX and IY, are all equal to the pivot string and are thus excluded from both sublists.
Lines 1650 to 1740: swap string1 and string2 descriptor blocks, a byte at a time, using implied addressing.
Lines 1760 to 1800: add three to 'sd1point' so as to point to the next string1 descriptor.
Lines 1820 to 1830: subtract three from 'sd2point' so as to point to the next string2 descriptor.
Lines 1880 to 1900: increment the array index IX, decrement the array index IY and unconditionally jump to 'first'.
Lines 1920 to 1930: increment the stackcounter ready for placing array index limits on the stack.
Lines 1960 to 2060: calculate the lengths of each sublist. Remember that after list/sublist partitioning that:

(a) 'lowhead' is always the current value stored at 'head'.
(b) 'lowtail' is the current value of the array index IY.
(c) 'highhead' is the current value of the array index IX.
(d) 'hightail' is always the current value stored at 'tail'.

The register pair DE holds the result of 'lowtail' – 'lowhead') and HL holds the result of ('hightail' – 'highhead'). Note that the quickest way of transferring the index register contents IX and IY to a register pair is a **PUSH xy** and a **POP rp.**
Lines 2090 to 2200: compare the two sublist lengths and push the array index limits, of the larger of the two, on the stack. The lower and upper limits of the smaller sublist are stored at 'head' and 'tail' respectively. An unconditional branch back to 'loop' is effected, whatever the outcome, so the next sublist can be partitioned.
Lines 2230 to 2250: check if any unsorted sublist limits remain on the stack waiting processing. If the stackcounter is zero then a forward branch to 'finish' is made.
Lines 2260 to 2280: decrement the stackcounter.
Lines 2300 to 2340: pop the unsorted sublist limits from the top of the stack in the reverse order to which they were pushed. These indices are stored in 'head' and 'tail' and a branch back to 'loop' forces another partitioning process.

Resident system extensions (RSX)

Subroutines can,if preferred, be called with a system command prefixed by a vertical bar. Instead of using the **CALL** command, followed by an execution address and parameter list, we can set up an RSX version which can be executed from BASIC or direct mode. For example, if the previous program was converted to an RSX we could have called it with:

|SORT,head%,tail%,@A$(0)

Notice that we do not need an execution address. We could call it with a command in much the same way as we would in BASIC. However, there are one or two snags. The designers of the Amstrad's software recommend that all RSX object code be self relocating so as to fall in with their dynamic allocation of memory. This is a rather complicated process so it will be sufficient here to treat the setting up of a number of example RSXs and postpone the more difficult concepts such as relocation and parameter passing until Chapter 13.

Logging on an RSX

The first step in setting up an RSX is to 'log it on' to the operating system's list of valid commands. They are called resident system extensions because they are, quite literally, extensions to the operating system repertoire. In order to log on a new RSX we need to use a firmware routine in the Kernal called **KL LOG EXT.** This can

be called, via the jumpblock, at address #BCD1. It has the following properties:

KL LOG EXT Call address **#BCD1**

Purpose: log on one or more RSX's to the operating system firmware. The routine must be called to add the RSX's name and address to the Kernal's list of external command servers.
Pre-call conditions: The register pair,BC, must be set to the address of the RSX's external command table. The HL register pair must be set to the address of an arbitrarily chosen, 4 byte, block of RAM. These locations are used by the Kernal as workspace. The Kernals storage area and the external command table must be in the central area of memory. That is to say, it must not lie 'beneath' a ROM.
Exit conditions: The register pair DE is corrupted but all other registers are preserved.

Logging on an RSX is complicated rather than difficult. Program 12.1 and its associated breakdown should help to unravel the details. The example RSX's themselves have been deliberately kept simple in order that the essential details are not obscured.

Program 12.3 Logging on an RSX

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                      10 ;DEMONSTRATION OF SETTING UP
                      20 ;RESIDENT SYSTEM EXTENSIONS (RSX)
8000                  30 begin:   EQU   #8000
8100                  40 top:     EQU   begin+#100
8000                  50          ORG   begin
                      60 ;Set up and CALL KL LOG EXT
8000  010A80          70          LD    BC,Ctable
8003  210081          80          LD    HL,top
8006  CDD1BC          90          CALL  #BCD1
8009  C9             100          RET
                     110 ;Set up external command table
800A  1580           120 Ctable: DEFW  Ntable
800C  C32780         130          JP    first
800F  C33180         140          JP    second
8012  C33B80         150          JP    third
                     160 ;Set up name table
8015  4C494E         170 Ntable: DEFM  "LIN"
8018  C5             180          DEFB  "E"+#80
8019  424C4F43       190          DEFM  "BLOCKLIN"
8021  C5             200          DEFB  "E"+#80
8022  42454C         210          DEFM  "BEL"
8025  CC             220          DEFB  "L"+#80
8026  00             230          DEFB  0
                     240 ;Draw line of 40 * ASCII(154)
8027  0628           250 first:  LD    B,#28
8029  3E9A           260          LD    A,#9A
```

```
802B  CD5ABB      270 loop:    CALL #BB5A
802E  10FB        280          DJNZ loop
8030  C9          290          RET
                  300 ;Draw line of 40 * ASCII(143)
8031  0628        310 second:  LD   B,#28
8033  3E8F        320          LD   A,#8F
8035  CD5ABB      330 loop2:   CALL #BB5A
8038  10FB        340          DJNZ loop2
803A  C9          350          RET
                  360 ;Ring BELL ASCII(7)
803B  3E07        370 third:   LD   A,7
803D  CD5ABB      380          CALL #BB5A
8040  C9          390          RET

Pass 2 errors: 00

Table used:    121   from    182
```

The example RSX commands logged on are as follows:

|LINE : displays 40 graphics characters of ASCII 154.
|BLOCKLINE : displays 40 graphics characters of ASCII 143.
|BELL : outputs the ASCII 7 control code for BELL (a beep).

The example cites only three commands but there can be as many, or as few, as you want. To log on the above commands to the external command server, type **CALL &7000** from BASIC. The firmware will then know, if any of the above commands are entered, where in memory, to find and execute the appropriate routine.

Breakdown of Program 12.3

Line 30: sets the assembly address at #7000 for the **ORG** assembler directive in line 50.
Line 40: sets the workspace area labelled 'top' at #7100.
Line 70: loads the register pair BC with the address of the external command table. We have labelled this 'Ctable' at line 120.
Line 80: loads the HL register pair with the address of the 4 byte workspace area demanded by the firmware routine **KL LOG EXT.** This is allocated at #7100 in the example, but any other 4 sequential locations that do not lie 'under' a ROM will do.
Line 90: calls the firmware routine **KL LOG EXT,** via the main jump block address #BCD1.
Line 100: returns control to the calling program. At this stage all the RSX routines will have been logged on and can be called by their above names. Remember to prefix them with a bar even if called from within a BASIC program.
Line 120: The firmware expects that the first two bytes of the external

command table will hold the address of the name table. The **DEFW** assembler directive evaluates its following expression, in this case the 2 byte address labelled 'Ntable', and copies it into memory from the address currently held in the program counter.
Lines 130 to 150: contain the associated jumps to the RSX routines. They are labelled 'first', 'second' and 'third'.
Lines 170 to 220: contain the name table and each entry corresponds, one for one, with its jump instruction in the external command table. Notice the format of the name table. The **DEFM** directive is used to copy the command name strings into memory up to, but excluding, the last character. The last character of the command name cannot be included with the **DEFM** string because it must have #80 added to its ASCII code for recognition by the firmware. The DEFB directive, which expects the following expression to evaluate to an 8 bit value, is used to perform the necessary arithmetic during assembly.
Line 230: acts as an end of name table marker (a zero data byte) and must always be included. On assembly the **DEFB** directive will place a zero data byte at the current address held in the program counter.
Lines 250 to 390: are the simple example RSX routines described above and are not intended to be of great practical use.

Summary

1 Parameters can be passed from machine code programs to machine code subroutines via registers, parameter blocks, fixed memory locations or the stack.
2 Parameters are passed from BASIC to machine code routine with the **CALL** command.
3 Algorithms programmed in machine code may not necessarily execute faster than more efficient algorithms written in BASIC.
4 On average, Quicksort is one of the fastest known sorting methods for large randomly ordered lists.
5 Occasionally, Quicksort may perform as bad as or worse than a bubble sort but it is unlikely in practice.
6 To alter the execution address of most assembler programs, the object code must be reassembled at the new address. Producing self relocating object code can be tedious but rewarding.
7 A BASIC string array consists of many string descriptors placed sequentially in memory.
8 A BASIC string descriptor consists of three bytes, a length byte followed by a 16 bit address.
9 The strings themselves are stored downwards in memory from HIMEM and are called the 'heap'.
10 The symbol '@' before a parameter in a CALL command indicates its address not its value.
11 Machine code subroutines, intended to be called from BASIC, are best loaded as a binary file from tape or disc.

12 An external command must be prefixed by a bar.
13 The **KL LOG EXT** firmware routine, called at #BCD1, must be used to log on an RSX.
14 An RSX should preferably be location independent or self relocatable.
15 Name table entries and command table jumps should correspond one to one.
16 The last character of a name table entry must have 80 hex added to its ASCII code.
17 The name table must be terminated with a zero data byte.
18 The first two bytes of the external command table must be the address of the name table.
19 The 4 byte workspace the firmware needs for RSX use, must be in the central area of RAM (not under a ROM).
20 The BC register pair must contain the external command table address. The HL register pair must contain the 4 byte workspace address before making a **CALL to KL LOG EXT.**

13 Self relocation of subroutines and resident system extensions

The more advanced requirements for setting up an RSX are described but the chapter is mainly concerned with self relocation procedures. As an example, we will convert the sorting program of the previous chapter into a self relocating RSX and show the various steps involved.

In addition, machine code sorting of BASIC two dimensional or rectangular arrays is treated. The rectangular array is a convenient format for database type programs which require the complete file to be resident in RAM and where fields are stored in one dimension and records in the other.

Dynamic allocation of memory

We noted in the previous chapter that the designers of the Amstrad recommend that any RSX is made location independent or *self relocating* in order to fall in with their dynamic allocation of memory. This is sound advice and applies to any machine code subroutine, whether an RSX or not. For example, as light pens, discs and graphics tablets are made available they may well grab memory from the top of the memory pool and thus push HIMEM downwards. We can never guarantee, on any machine, which is the highest usable memory address. Obviously, if we err on the safe side, and assemble object code at a low, but safe address, we will waste a lot of memory. On the other hand, if we neglect this problem, our subroutine or RSX may not work on some machines due to conflicts of memory requirements.

Self relocation of object code

When developing a large stand-alone machine code program, assembled in the lower regions of memory, the above warnings are not so important since it is unlikely that it will use memory in the upper regions of the memory pool. However, when using machine

code subroutines, or RSX's, intended to be called from BASIC then self relocation of object code is certainly desirable. In this way the machine code bytes can be loaded as a binary file from tape or disc and placed immediately under the current value of HIMEM, thus maximising the available memory pool for BASIC. Before logging on the RSX the machine code routine can be arranged to modify itself for execution at its load address without re-assembly.

Location independent object code

It is possible, in short simple subroutines or RSX's, to use a restricted
set of instructions which enables the assembled object code execute in any chosen region of memory. The following advice is given as a guide to producing location independent object code.

(a) Use registers and the stack rather than fixed memory locations to manipulate and store variables.

(b) Use *program relative* addressing. In other words choose relative jump instructions rather than direct jumps. If the operand exceeds the limit of one byte then use two or more relative jumps such as:

```
loop      :
          :
          :
stage1    : JR Z,loop
          :
          :
          : JR Z,stage1
```

(c) Do not use subroutines which are called with a 16 bit address. This means the CALL instruction is not really suitable.

In a larger subroutine or RSX it may not be desirable or feasible to restrict ourselves to the above rules. Heavy use of the stack, in particular, can lead to a lot of untidy and inefficient code. In short, subroutines written in this way are difficult to read and, more importantly, to debug or modify. Where location independence is not feasible we must resort to a self relocation method.

Self relocation problems

Before getting involved in the details, let's take a look at what needs to be altered in the average machine code program so that it will always execute correctly, irrespective of where it is stored in memory:

1 All operands included in absolute jump instructions because they all possess a 16 bit absolute address.
2 All instructions which have an operand consisting of a fixed memory reference. These include labelled addresses for variable data.
3 All CALL instruction operands. These are all 16 bit absolute addresses.

To elaborate on item 2 above there does not appear to be any region, other than in the main memory pool, that can freely be used for the temporary storage of variable data. Even if a region were to be found in the BASIC or operating system workspace we can never be certain that the area will not be utilised in some future redesign work or modification. It is far better to be in complete control and allow the RSX and its associated workspace to be included in the relocation process.

Another problem is that instructions may have different lengths. For example look at the representative sample in the following table.

Instruction or assembler directive	*Op-code length*	*Operand length or data bytes*
LD A,(label)	1	2
DEFW label	–	2
LD HL,(label)	1	2
LD rp, (label)	2	2
JP C,label	1	2
JP label	1	2

For more detail on instructions and their lengths refer to Appendix 1 and Appendix 4.

When altering locations in the program we must be careful that we change only the operands and not the Op-code.

Converting a subroutine to an RSX

To convert a subroutine to its self relocating RSX version is tedious but fairly straightforward. Program 13.1 is the RSX version of Program 12.1 and shows all the changes that are necessary to the source code listing in order to produce self relocating object code. Since the routine is fairly long and consists of a large selection of instructions, most eventualities likely to occur in your own programs will have been covered by way of example.

Program 13.1 String Quicksort as an RSX

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                        10 ;QUICKSORT
                        20 ;OF A STRING ARRAY
                        30 ;(LOCATION INDEPENDENT RSX)
7000                    40 begin:  EQU  #7000
7203                    50 top:    EQU  begin+#0203
7203                    60 tail:   EQU  top
7205                    70 head:   EQU  top+2
7207                    80 array:  EQU  top+4
7209                    90 stackc: EQU  top+6
720A                   100 plen:   EQU  top+7
720B                   110 ppoint: EQU  top+8
720D                   120 sd1poi: EQU  top+10
720F                   130 sd2poi: EQU  top+12
E9E1                   140 code:   EQU  #E9E1
0030                   150 loc:    EQU  #0030
7000                   160         ORG  begin
                       170 ;Program user restart RST #6
                       180 ;to get address of ref in HL
7000  21E1E9           190         LD   HL,code
7003  223000           200         LD   (loc),HL
                       210 ;Call the restart routine
                       220 ;placed at location #30
7006  F7               230         RST  loc
7007  44               240 ref:    LD   B,H
7008  4D               250         LD   C,L
7009  218E01           260         LD   HL,Ltable-ref
700C  09               270         ADD  HL,BC
                       280 ;Pick up data from Ltable.
                       290 ;Make absolute by adding ref.
                       300 ;Access locations, add ref
                       310 ;to the relative values present
700D  5E               320 alter:  LD   E,(HL)
700E  23               330         INC  HL
700F  56               340         LD   D,(HL)
7010  23               350         INC  HL
7011  7A               360         LD   A,D
7012  B3               370         OR   E
7013  2811             380         JR   Z,end
7015  E5               390         PUSH HL
7016  EB               400         EX   DE,HL
7017  09               410         ADD  HL,BC
7018  E5               420         PUSH HL
7019  5E               430         LD   E,(HL)
701A  23               440         INC  HL
701B  56               450         LD   D,(HL)
701C  EB               460         EX   DE,HL
701D  09               470         ADD  HL,BC
701E  EB               480         EX   DE,HL
701F  E1               490         POP  HL
7020  73               500         LD   (HL),E
7021  23               510         INC  HL
7022  72               520         LD   (HL),D
```

```
 7023  E1         530          POP  HL
 7024  18E7       540          JR   alter
                  550 ;Set up and CALL KL LOG EXT
 7026  012900     560 end:     LD   BC,Ctable-ref



Hisoft GENA3.1 Assembler. Page     2.

7029  218A01      570 L10:     LD   HL,Wspace-ref
702C  CDD1BC      580          CALL #BCD1
702F  C9          590          RET
                  600 ;Set up external command table
7030  2E00        610 Ctable:  DEFW Ntable-ref
7032  C33600      620 L20:     JP   sort-ref
                  630 ;Set up name table
7035  534F5254    640 Ntable:  DEFM "SORTST"
703B  D2          650          DEFB "R"+#80
703C  00          660          DEFB 0
                  670 ;Pick up and store base address
                  680 ;of string descriptors : array
703D  DD6E00      690 sort:    LD   L,(IX)
7040  DD6601      700          LD   H,(IX+1)
7043  220002      710 L30:     LD   (array-ref),HL
                  720 ;Pick up tail of array : tail
7046  DD6E02      730          LD   L,(IX+2)
7049  DD6603      740          LD   H,(IX+3)
704C  22FC01      750 L40:     LD   (tail-ref),HL
                  760 ;Pick up head of array : head
704F  DD6E04      770          LD   L,(IX+4)
7052  DD6605      780          LD   H,(IX+5)
7055  22FE01      790 L50:     LD   (head-ref),HL
                  800 ;Set stackcounter to zero
7058  97          810          SUB  A
7059  320202      820 L60:     LD   (stackcount-ref),A
                  830 ;Branch to bypass if head>=tail
705C  2AFE01      840 loop:    LD   HL,(head-ref)
705F  ED5BFC01    850 L70:     LD   DE,(tail-ref)
7063  A7          860          AND  A
7064  ED52        870          SBC  HL,DE
7066  D27301      880 L80:     JP   NC,bypass-ref
                  890 ;Initialise highhead and lowtail
                  900 ; IX & IY respectively
7069  DD2AFE01    910 L90:     LD   IX,(head-ref)
706D  FD2AFC01    920 L100:    LD   IY,(tail-ref)
                  930 ;Calculate pivot string
                  940 ;descriptor address : HL
7071  2AFE01      950 L110:    LD   HL,(head-ref)
7074  ED5BFC01    960 L120:    LD   DE,(tail-ref)
7078  19          970          ADD  HL,DE
7079  CB3C        980          SRL  H
707B  CB1D        990          RR   L
707D  54         1000          LD   D,H
707E  5D         1010          LD   E,L
707F  29         1020          ADD  HL,HL
7080  19         1030          ADD  HL,DE
7081  ED4B0002   1040 L130:    LD   BC,(array-ref)
```

```
7085  09        1050          ADD  HL,BC
                1060 ;Get length and address of
                1070 ;pivot string : plen & ppoint
7086  7E        1080          LD   A,(HL)
7087  320302    1090 L140:    LD   (plen-ref),A
708A  23        1100          INC  HL
708B  5E        1110          LD   E,(HL)
708C  23        1120          INC  HL
708D  56        1130          LD   D,(HL)
708E  ED530402  1140 L150:    LD   (ppoint-ref),DE


Hisoft GENA3.1 Assembler. Page     3.


                1150 ;Set pointer to first string
                1160 ;descriptor: sd1point
7092  2AFE01    1170 L160:    LD   HL,(head-ref)
7095  54        1180          LD   D,H
7096  5D        1190          LD   E,L
7097  29        1200          ADD  HL,HL
7098  19        1210          ADD  HL,DE
7099  09        1220          ADD  HL,BC
709A  220602    1230 L170:    LD   (sd1point-ref),HL
                1240 ;Set pointer to second string
                1250 ;descriptor: sd2point
709D  2AFC01    1260 L180:    LD   HL,(tail-ref)
70A0  54        1270          LD   D,H
70A1  5D        1280          LD   E,L
70A2  29        1290          ADD  HL,HL
70A3  19        1300          ADD  HL,DE
70A4  09        1310          ADD  HL,BC
70A5  220802    1320 L190:    LD   (sd2point-ref),HL
                1330 ;Get length and address of first
                1340 ;string: C & DE
70A8  2A0602    1350 first:   LD   HL,(sd1point-ref)
70AB  4E        1360          LD   C,(HL)
70AC  23        1370          INC  HL
70AD  5E        1380          LD   E,(HL)
70AE  23        1390          INC  HL
70AF  56        1400          LD   D,(HL)
                1410 ;Compare first string to pivot
                1420 ;branch to proc1 if string<pivot
70B0  0600      1430          LD   B,0
70B2  2A0402    1440 L200:    LD   HL,(ppoint-ref)
70B5  1A        1450 comp:    LD   A,(DE)
70B6  BE        1460          CP   (HL)
70B7  3811      1470          JR   C,proc1
70B9  201C      1480          JR   NZ,second
70BB  04        1490          INC  B
70BC  3A0302    1500 L210:    LD   A,(plen-ref)
70BF  B8        1510          CP   B
70C0  2815      1520          JR   Z,second
70C2  79        1530          LD   A,C
70C3  B8        1540          CP   B
70C4  2804      1550          JR   Z,proc1
70C6  13        1560          INC  DE
70C7  23        1570          INC  HL
```

```
70C8  20EB       1580          JR   NZ,comp
                 1590 ;Add 3 to sd1point
70CA  2A0602     1600 proc1:   LD   HL,(sd1point-ref)
70CD  23         1610          INC  HL
70CE  23         1620          INC  HL
70CF  23         1630          INC  HL
70D0  220602     1640 L220:    LD   (sd1point-ref),HL
                 1650 ;increment highhead
70D3  DD23       1660          INC  IX
70D5  18D1       1670          JR   first
                 1680 ;Get length and address of
                 1690 ;second string : C & DE
70D7  2A0802     1700 second:  LD   HL,(sd2point-ref)
70DA  4E         1710          LD   C,(HL)
70DB  23         1720          INC  HL


Hisoft GENA3.1 Assembler. Page     4.

70DC  5E         1730          LD   E,(HL)
70DD  23         1740          INC  HL
70DE  56         1750          LD   D,(HL)
                 1760 ;Compare second string to pivot
                 1770 ;branch to proc2 if string>pivot
70DF  0600       1780          LD   B,0
70E1  EB         1790          EX   DE,HL
70E2  ED5B0402   1800 L230:    LD   DE,(ppoint-ref)
70E6  1A         1810 comp2:   LD   A,(DE)
70E7  BE         1820          CP   (HL)
70E8  3811       1830          JR   C,proc2
70EA  201C       1840          JR   NZ,over
70EC  04         1850          INC  B
70ED  79         1860          LD   A,C
70EE  B8         1870          CP   B
70EF  2817       1880          JR   Z,over
70F1  3A0302     1890 L240:    LD   A,(plen-ref)
70F4  B8         1900          CP   B
70F5  2804       1910          JR   Z,proc2
70F7  13         1920          INC  DE
70F8  23         1930          INC  HL
70F9  20EB       1940          JR   NZ,comp2
                 1950 ;Subtract 3 from sd2point
70FB  2A0802     1960 proc2:   LD   HL,(sd2point-ref)
70FE  2B         1970          DEC  HL
70FF  2B         1980          DEC  HL
7100  2B         1990          DEC  HL
7101  220802     2000 L250:    LD   (sd2point-ref),HL
                 2010 ;decrement lowtail
7104  FD2B       2020          DEC  IY
7106  18CF       2030          JR   second
                 2040 ;Compare sd1point to sd2point
                 2050 ;Br. proc3 if sd1point<sd2point
                 2060 ;Br. skip if sd1point>sd2point
                 2070 ;if = dec lowtail & inc highhead
7108  A7         2080 over:    AND  A
7109  2A0602     2090 L260:    LD   HL,(sd1point-ref)
710C  ED5B0802   2100 L270:    LD   DE,(sd2point-ref)
```

```
7110  ED52      2110          SBC  HL,DE
7112  3808      2120          JR   C,proc3
7114  202D      2130          JR   NZ,skip
7116  FD2B      2140          DEC  IY
7118  DD23      2150          INC  IX
711A  1827      2160          JR   skip
                2170 ;swop string descriptors
711C  0603      2180 proc3:   LD   B,3
711E  2A0602    2190 L280:    LD   HL,(sd1point-ref)
7121  1A        2200 swop:    LD   A,(DE)
7122  4E        2210          LD   C,(HL)
7123  EB        2220          EX   DE,HL
7124  71        2230          LD   (HL),C
7125  12        2240          LD   (DE),A
7126  13        2250          INC  DE
7127  23        2260          INC  HL
7128  10F7      2270          DJNZ swop
                2280 ;Add 3 to sd1point
712A  2A0602    2290 L290:    LD   HL,(sd1point-ref)
712D  23        2300          INC  HL


Hisoft GENA3.1 Assembler. Page     5.


712E  23        2310          INC  HL
712F  23        2320          INC  HL
7130  220602    2330 L300:    LD   (sd1point-ref),HL
                2340 ;subtract 3 from sd2point
7133  2A0802    2350 L310:    LD   HL,(sd2point-ref)
7136  2B        2360          DEC  HL
7137  2B        2370          DEC  HL
7138  2B        2380          DEC  HL
7139  220802    2390 L320:    LD   (sd2point-ref),HL
                2400 ;dec highhead:inc lowtail
713C  DD23      2410          INC  IX
713E  FD2B      2420          DEC  IY
7140  C3A100    2430 L330:    JP   first-ref
                2440 ;increment stackcounter
7143  210202    2450 skip:    LD   HL,stackcount-ref
7146  34        2460          INC  (HL)
                2470 ;Calc lowtail-lowhead &
                2480 ; hightail-highead
7147  A7        2490          AND  A
7148  FDE5      2500          PUSH IY
714A  E1        2510          POP  HL
714B  ED5BFE01  2520 L340:    LD   DE,(head-ref)
714F  ED52      2530          SBC  HL,DE
7151  EB        2540          EX   DE,HL
7152  A7        2550          AND  A
7153  2AFC01    2560 L350:    LD   HL,(tail-ref)
7156  DDE5      2570          PUSH IX
7158  C1        2580          POP  BC
7159  ED42      2590          SBC  HL,BC
                2600 ;Compare results & stack larger
                2610 ;limits : process smaller limits
715B  A7        2620          AND  A
715C  ED52      2630          SBC  HL,DE
```

```
715E  300D      2640          JR   NC,sthigh
7160  2AFE01    2650 L360:    LD   HL,(head-ref)
7163  E5        2660          PUSH HL
7164  FDE5      2670          PUSH IY
7166  DD22FE01  2680 L370:    LD   (head-ref),IX
716A  C35500    2690 L380:    JP   loop-ref
716D  DDE5      2700 sthigh:  PUSH IX
716F  2AFC01    2710 L390:    LD   HL,(tail-ref)
7172  E5        2720          PUSH HL
7173  FD22FC01  2730 L400:    LD   (tail-ref),IY
7177  C35500    2740 L410:    JP   loop-ref
                2750 ;compare stackcounter to zero
717A  3A0202    2760 bypass:  LD   A,(stackcount-ref)
717D  FE00      2770          CP   0
717F  280F      2780          JR   Z,finish
                2790 ;decrement stackcounter
7181  3D        2800          DEC  A
7182  320202    2810 L420:    LD   (stackcount-ref),A
                2820 ;Pop tail & head from stack
7185  E1        2830          POP  HL
7186  22FC01    2840 L430:    LD   (tail-ref),HL
7189  E1        2850          POP  HL
718A  22FE01    2860 L440:    LD   (head-ref),HL
718D  C35500    2870 L450:    JP   loop-ref
7190  C9        2880 finish:  RET


Hisoft GENA3.1 Assembler. Page     6.


7191            2890 Wspace:  DEFS 4
7195  2000      2900 Ltable:  DEFW end+1-ref
7197  2900      2910          DEFW Ctable-ref
7199  5600      2920          DEFW loop+1-ref
719B  A200      2930          DEFW first+1-ref
719D  C400      2940          DEFW proc1+1-ref
719F  D100      2950          DEFW second+1-ref
71A1  F500      2960          DEFW proc2+1-ref
71A3  3D01      2970          DEFW skip+1-ref
71A5  7401      2980          DEFW bypass+1-ref
71A7  23002C00  2990          DEFW L10+1-ref,L20+1-ref
71AB  3D004600  3000          DEFW L30+1-ref,L40+1-ref
71AF  4F005300  3010          DEFW L50+1-ref,L60+1-ref
71B3  5A006000  3020          DEFW L70+2-ref,L80+1-ref
71B7  64006800  3030          DEFW L90+2-ref,L100+2-ref
71BB  6B006F00  3040          DEFW L110+1-ref,L120+2-ref
71BF  7C008100  3050          DEFW L130+2-ref,L140+1-ref
71C3  89008C00  3060          DEFW L150+2-ref,L160+1-ref
71C7  94009700  3070          DEFW L170+1-ref,L180+1-ref
71CB  9F00AC00  3080          DEFW L190+1-ref,L200+1-ref
71CF  B600CA00  3090          DEFW L210+1-ref,L220+1-ref
71D3  DD00EB00  3100          DEFW L230+2-ref,L240+1-ref
71D7  FB000301  3110          DEFW L250+1-ref,L260+1-ref
71DB  07011801  3120          DEFW L270+2-ref,L280+1-ref
71DF  24012A01  3130          DEFW L290+1-ref,L300+1-ref
71E3  2D013301  3140          DEFW L310+1-ref,L320+1-ref
71E7  3A014601  3150          DEFW L330+1-ref,L340+2-ref
71EB  4D015A01  3160          DEFW L350+1-ref,L360+1-ref
```

```
71EF  61016401  3170        DEFW L370+2-ref,L380+1-ref
71F3  69016E01  3180        DEFW L390+1-ref,L400+2-ref
71F7  71017C01  3190        DEFW L410+1-ref,L420+1-ref
71FB  80018401  3200        DEFW L430+1-ref,L440+1-ref
71FF  87010000  3210        DEFW L450+1-ref,0

Pass 2 errors: 00

Table used:    901  from  1000
```

The first thing we should notice about Program 13.1 over its predecessor is the extra block of instructions at the head of the listing (lines 190 to 680). Besides setting up the routine as an RSX, there are extra sections of code for changing key addresses and picking up the addresses where the object code itself is loaded.

Converting existing code to a self relocatable RSX

Programs which already exist, such as Program 12.1, can be converted to self relocatable form by means of the following steps:

Step 1: Program the user restart RST 6
The user restart, RST 6, is a section of memory available to the user for extending the instruction set of the Z80A. It can be visualised as a small subroutine called by the single byte instruction, **RST #30**. The number of locations available for the user restart are restricted to the 8 byte address range #0030 to #0037.

Accordingly, the machine code program will need to know where in memory it has been loaded so that it can change its key locations. We could, of course, simply POKE in these values at a location which is a fixed offset from the load address. This would be a nuisance, especially to first time users, so a fully automatic method using RST 6 is preferred.

When RST 6 is executed by a **RST #30** instruction, the return address is first placed on the stack. So if we program RST 6 as follows:

POP HL
JP (HL)

we can withdraw the return address from the stack into the HL register pair and use a register indirect jump back to the return address. This will leave the address of the location, immediately following the **RST #30** instruction, in the HL register pair. This is an ideal reference point from which to alter the key addresses necessary for relocation. Consequently, in Program 13.1 we have labelled this location 'ref'. The breakdown of the relevant parts of Program 13.1 are as follows:

Line 140: assigns the op-code bytes of **POP HL** and **JP (HL)** to the label 'code'. These are #E1 and #E9 respectively. The bytes are shown in reverse order in the listing because a future load into HL will place E1 in register L and E9 in register H.
Line 150: assigns the value #0030 to the location 'loc'. This is the address of the user restart, RST 6.
Lines 190 to 200: load the above op-code bytes into the restart locations, #30 and #31.
Line 230: calls the use restart with **RST loc.**
Line 240: the instruction in this line has the label 'ref'. The address is in the HL register pair at this point.

Step 2: Convert absolute addresses to 'ref' relative
When an instruction is found that has an absolute 16 bit address in its operand, subtract 'ref' from it to make the address relative to the location with that label. At the same time, if the label field of the assembler is empty, place numbered markers such as those in Program 13.1. There is no need to do this if a label is already present. When labelling, always go up in steps of ten or so in case you miss one out.

As an example, we will compare the equivalent lines of Program 12.1 and 13.1, lines 430 and 960 respectively.

In program 12.1 the line is:

```
7037   ED5B0372   430             LD DE,(tail)
```

In program 13.1 the equivalent line is changed to:

```
7074   ED5BFC01   960 L120:       LD DE,(tail-ref)
```

The points worthy of mention are:

(a) The absolute address label 'tail' has been changed to 'tail-ref'.
(b) The instruction is marked by a label 'L120' in the label field of the assembler.
(c) The assembler has performed the arithmetic on the address 'tail-ref' expression and has placed the relative address, offset from 'ref', in the object code field. The absolute address #7203 has been changed to the 'ref' relative address of #01FC in the assembly process.

It is important to stress, once again, that you save your source code on tape or disc before executing the object code program.

Step 3: Set up a location pointer table
All instructions using 16 bit absolute addresses should at this stage be marked off with a label. The location pointer table should hold, as 2 byte data words, the 'ref' relative addresses of the operands to be modified. Expressions following the DEFW directive are used to calculate the addresses. The location pointer table occupies lines

2900 to 3210 of Program 13.1 and has the label 'Ltable' in line 2900. There are three different types of expression each corresponding to the different lengths of instruction.

At the moment the labels L10, L20, L30 etc index the op-code addresses, not the operand addresses. To recitify this, we need to add-in a non constant offset, depending on op-code length (if any) so that only the operand addresses which need to be changed are present in the table.

Each entry into the table is an expression which evaluates to a 'ref' relative 16 bit address. These expression can be either on the same line separated by commas (lines 2990 to 3210), or else in a single line (lines 2900 to 2980). Notice that lines 2900 to 2980 use labels other than those numbered because the locations were already labelled for other reasons. It is a good idea to include them at the head of the table.

To illustrate the setting of the table's DEFW expressions, the three different types occurring in Program 13.1 are described below.

Line 2910: The location labelled 'Ctable' (refer to line 610) does not contain an actual instruction. It is simply the address data of the RSX name table. In this case to make it 'ref' relative we simply subtract 'ref'.

Line 3070: The first address expression on this line is

'L170+1−ref'.

Looking back at the instruction labelled 'L170' in line 1230, we find that the instruction is of the form

LD (addr),HL

which is a three byte instruction. The op-code is one byte long and the 16 bit address operand follows it. This is why the offset of one is added. Again the address value is made 'ref' relative by subtracting 'ref'.

Line 3100: The first address expression is

'L230+2−ref'.

Looking back at the instruction labelled 'L230' in line 1800, we find that it is of the form

LD rp,(addr)

which is a four byte instruction. This time the op-code extends over two bytes so the address operand is offset by 2 from the label 'L230', hence the expression

'L230+2−ref'.

Finally, a zero data word (**DEFW 0**) is used as a marker to denote the end of the table.

Step 4: Add a location change routine
This routine is needed to pick up each relative address from the table and convert it to absolute form by adding back 'ref'. The corresponding locations are then accessed and their relative contents made absolute again by the addition of 'ref'. Remember that the new value of 'ref' will have been determined for the new execution address by the RST #30 call described in Step 1. The routine, used in Program 13.1, lies between lines 240 to 540. The breakdown of this routine is as follows.
Lines 240 to 250: the current value of the address 'ref' will be present in the HL register pair. For later use, it is copied into the BC register pair.
Lines 260 to 270: the 'ref' relative address of the location pointer table is loaded into the HL register pair. The contents are made absolute by adding BC, thus initialising HL as a data pointer to the table.
Lines 320 to 350: the currently accessed location table value is loaded into the DE register pair, using implied addressing.
Lines 360 to 380: a check is made for the end of table marker which is a data word set to #0000. If DE is found to contain zero then the loop is terminated by a branch to 'end'.
Line 390: the data pointer, HL, is temporarily placed on the stack.
Line 400 to 420: the current relative address in DE is swapped over to HL, and BC added to convert it to absolute. A copy of the absolute address pointer value, left in HL, is placed on the stack for later re-use.
Lines 430 to 450: the operand, of the instruction to be altered is loaded into the DE register pair using implied addressing.
Lines 460 to 480: this value is made absolute by adding the value of 'ref', still in BC, to the DE register pair.
Lines 490 to 520: the address pointer HL is restored from the stack and the modified contents, in DE, are written back in the instruction operand using implied addressing.
Lines 530 to 540: the table data pointer is restored from the stack and an unconditional branch is made back to 'alter' so that the process can be repeated on the next table entry.

Step 5: Log on the RSX
This section is placed from line 560 to line 660 of Program 13.1 and is similar in form to that previously decribed for Program 12.3. The only differences are:

1 Any references to absolute addresses in operands are subject to the above guidelines.

2 The 4 byte workspace, specified for the **KL LOG EXT (#BCD1)** call, is a labelled location 'Wspace'. Line 2890 shows that 4 bytes are reserved using the DEFS assembler directive starting at this address.

3 'SORTSTR' is the only external command name logged on.

Loading and testing the string sort RSX

Once assembled at &7000, the object code can be saved to tape or disc as a binary file. The RSX can then be loaded back into RAM at any address within the memory pool. Once in place, it is essential to call it at its chosen load address with the following command:

CALL <load adddress>

The reason for this is twofold:

1 To log on the RSX to the external command server.
2 To alter certain instruction operands for execution at the load address.

Note that, at this stage, only the code necessary for intitialisation and relocation is executed. The RSX itself, is subsequently executed each time its external command name is recognised. A suitable RSX loader, written in BASIC, occupies lines 20 to 60 of Program 13.2. However, if preferred, the RSX can be automatically loaded directly beneath HIMEM. The load address can be determined from the expression (HIMEM – L – 1) where L is the length of the RSX in bytes including workspace.

Lines 70 to 400 of Program 13.2 are simply a test routine which sets up a random string array, executes the RSX and displays the sorted array.

Program 13.2 Loader/Test program for the string sort RSX

```
10 REM TEST PROGRAM: STRING SORT (RSX)
20 CLS
30 INPUT"Load address";addr
40 MEMORY ABS(addr)-1
50 LOAD "SORTSTR.BIN",addr
60 CALL addr
70 CLS
80 INPUT"Sort how many strings";NUMBER%
90 PRINT
100 REM FILL AND DISPLAY RANDOM ARRAY
110 DIM A$(NUMBER%)
120 head%=1:tail%=NUMBER%
130 FOR R%=head% TO tail%
140 B$=""
150 A%=6*RND+1
160 FOR Z%=1 TO A%
170 N%=25*RND
180 K$=CHR$(N%+65)
190 B$=B$+K$
200 NEXT
210 A$(R%)=B$
220 PRINT A$(R%)
230 NEXT
240 PRINT:PRINT
250 PRINT"SORTING ARRAY"
260 PRINT:PRINT
```

```
270 START=TIME/300
280 |SORTSTR,head%,tail%,@A$(0)
290 T=TIME/300-START
300 FOR R%=head% TO tail%
310 PRINT A$(R%)
320 NEXT
330 PRINT
340 PRINT"STRINGS SORTED=";NUMBER%
350 PRINT
360 PRINT"SORTING TIME=";ROUND(T,2);"SECONDS"
370 PRINT
380 INPUT"Another test (Y/N)";K$
390 K$=UPPER$(K$)
400 IF K$="Y" THEN ERASE A$:GOTO 70 ELSE END
```

Producing the RSX binary file
The RSX binary file can be produced by the following method:

(a) Load the HiSoft DEVPAC assembler at, say, address &2000 hex.
(b) Type in the assembly code listing, Program 13.1.
(c) Save a copy of the source code on tape (or disc) by typing:

P 10,3210,SORTSTR.RSX

The source code file can be reloaded at some later time with

G,,SORTSTR.RSX

(d) Assemble the code by typing the assembler command, **A.** When asked for table size, respond with 1000.
(e) Clear any typing errors reported.
(f) Save the object code as a binary file by typing the assembler command:

O,,SORTSTR.BIN

(g) Perform a hard reset to clear memory.
(h) Type in and save on disc the test program, Program 13.2. This will automatically load, run and test the RSX at the chosen load address. Remember to save the program on tape or disc before a RUN.

Calling the RSX from BASIC

In the previous Chapter we executed the non-relocatable version of Quicksort, Program 12.1 with the following command:

CALL &7000,head%,tail%,@A$(0)

We did not need, as we do for the RSX version, an initialisation call to either log on the RSX or change any location contents. Nevertheless, the method, content and mechanism of parameter passing described there are equally valid for the self relocating RSX version. The only difference is that we execute it with the following, more convenient, variant:

|SORTSTR,head%,tail%,@A$(0)

RSX for sorting rectangular arrays

Program 13.3 is a self relocating RSX which is capable of sorting rectangular arrays. It is fast. In fact, it will sort a computerful of records in a second. The routine can sort records with up to 85 fields which is a limit well above the demands of most files. Before describing the routine, we will examine what rectangular arrays are, what they can be used for and how they are stored in memory.

RAM based data files and rectangular arrays

A data organisation, commonly encountered in RAM based filing systems, is the two-dimensional or rectangular string array. For example, if a RAM based file is DIMensioned A$(FIELDS%, RECORDS%) then the data array A$ can be considered to contain RECORDS% records each of FIELDS% fields. We can define any field in a specific record by A$(F%,R%) where F% is the field number and R% is the record number. The following table shows how to visualise such a file in memory.

Field headings	*Field 0* *A$(0,0)*	*Field 1* *A$(1,0)*	*Field 2* *A$(2,0)*
Record 1	A$(0,1)	A$(1,1)	A$(2,1)
Record 2	A$(0,2)	A$(1,2)	A$(2,2)
Record 3	A$(0,3)	A$(1,3)	A$(2,3)
Record 4	A$(0,4)	A$(1,4)	A$(2,4)
Record 5	A$(0,5)	A$(1,5)	A$(2,5)
Record n	A$(0,n)	A$(1,n)	A$(2,n)

Using rectangular arrays effectively

When BASIC DIMensions a rectangular array, 3 byte string descriptors are reserved for each array element whether they are used

or not. Unfortunately, this includes all zero indexed elements. In order to maximise the use of available memory we should ensure that field zero is used as one of the legitimate fields. If we do this then we can DIMension, say, a 400 record, 5 field data file as A$(4,400) rather than A$(5,400) which may, at first sight, seem an obvious choice. Neglect of this point could waste 400*3=1200 bytes of memory unnecessarily.

How BASIC organises a rectangular array in memory

If records are to be sorted according to a specific field, we need to modify our routines to exchange all fields of the record pairs each time.

The sequential order in which the BASIC interpreter stores the 3 byte string descriptors are as follows.

A$(0,0) A$(1,0) A$(2,0) A$(0,1) A$(1,1) A$(2,1) – – – – – A$(0,n) A$(1,n) A$(2,n)

The second index, or dimension, is the one with sequential integrity in memory. This is the reason why the field index is chosen first and the record index second. The record's string descriptors occupy a sequential block of memory and thus can be moved about efficiently using machine code. Access to the string descriptor of a particular field is a simple case of adding an offset, equal to a fixed multiple of three, to the zeroth string descriptor of each record.

Program 13.3 Quicksort of a rectangular string array (RSX)

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                       10 ;QUICKSORT OF A RECTANGULAR
                       20 ;STRING ARRAY
                       30 ;(LOCATION INDEPENDENT RSX)
7000                   40 begin:  EQU  #7000
7270                   50 top:    EQU  begin+#270
7270                   60 tail:   EQU  top
7272                   70 head:   EQU  top+2
7274                   80 array:  EQU  top+4
7276                   90 stackc: EQU  top+6
7277                  100 plen:   EQU  top+7
7278                  110 ppoint: EQU  top+8
727A                  120 sd1poi: EQU  top+10
727C                  130 sd2poi: EQU  top+12
727E                  140 bytes:  EQU  top+14
7280                  150 offset: EQU  top+16
E9E1                  160 code:   EQU  #E9E1
```

```
0030                170 loc:    EQU  #0030
7000                180         ORG  begin
                    190 ;Program user restart RST #6
                    200 ;to return address of ref in HL
7000  21E1E9        210         LD   HL,code
7003  223000        220         LD   (loc),HL
                    230 ;Call the restart routine
                    240 ;placed at location #30
7006  F7            250         RST  loc
7007  44            260 ref:    LD   B,H
7008  4D            270         LD   C,L
7009  21DD01        280         LD   HL,Ltable-ref
700C  09            290         ADD  HL,BC
                    300 ;Pick up data from Ltable.
                    310 ;Make absolute by adding ref.
                    320 ;Access locations, add ref
                    330 ;to the relative values present.
700D  5E            340 alter:  LD   E,(HL)
700E  23            350         INC  HL
700F  56            360         LD   D,(HL)
7010  23            370         INC  HL
7011  7A            380         LD   A,D
7012  B3            390         OR   E
7013  2811          400         JR   Z,end
7015  E5            410         PUSH HL
7016  EB            420         EX   DE,HL
7017  09            430         ADD  HL,BC
7018  E5            440         PUSH HL
7019  5E            450         LD   E,(HL)
701A  23            460         INC  HL
701B  56            470         LD   D,(HL)
701C  EB            480         EX   DE,HL
701D  09            490         ADD  HL,BC
701E  EB            500         EX   DE,HL
701F  E1            510         POP  HL
7020  73            520         LD   (HL),E
7021  23            530         INC  HL
7022  72            540         LD   (HL),D
7023  E1            550         POP  HL
7024  18E7          560         JR   alter


Hisoft GENA3.1 Assembler. Page     2.

                    570 ;Set up and CALL KL LOG EXT
7026  012900        580 end:    LD   BC,Ctable-ref
7029  21D901        590 L10:    LD   HL,Wspace-ref
702C  CDD1BC        600         CALL #BCD1
702F  C9            610         RET
                    620 ;Set up external command table
7030  2E00          630 Ctable: DEFW Ntable-ref
7032  C33700        640 L20:    JP   sort-ref
                    650 ;Set up name table
7035  534F5254      660 Ntable: DEFM "SORT2ST"
703C  D2            670         DEFB "R"+#80
703D  00            680         DEFB 0
                    690 ; Initialise locations
```

```
703E  97        700 sort:   SUB  A
703F  327A02    710 L30:    LD   (offset+1-ref),A
7042  327B02    720 L40:    LD   (bytes+1-ref),A
7045  326F02    730 L50:    LD   (stackcount-ref),A
                740 ;Pick up and store base address
                750 ;of string descriptors : array
7048  DD6E00    760         LD   L,(IX)
704B  DD6601    770         LD   H,(IX+1)
704E  226D02    780 L60:    LD   (array-ref),HL
                790 ;Pick up field sort index
                800 ;multiply by three: offset
7051  DD7E02    810         LD   A,(IX+2)
7054  47        820         LD   B,A
7055  CB27      830         SLA  A
7057  80        840         ADD  A,B
7058  327902    850 L70:    LD   (offset-ref),A
                860 ;Pick up number of fields, add 1
                870 ;then multiply by three : bytes
705B  DD7E04    880         LD   A,(IX+4)
705E  3C        890         INC  A
705F  47        900         LD   B,A
7060  CB27      910         SLA  A
7062  80        920         ADD  A,B
7063  327702    930 L80:    LD   (bytes-ref),A
                940 ;Pick up tail of array : tail
7066  DD6E06    950         LD   L,(IX+6)
7069  DD6607    960         LD   H,(IX+7)
706C  226902    970 L90:    LD   (tail-ref),HL
                980 ;Pick up head of array : head
706F  DD6E08    990         LD   L,(IX+8)
7072  DD6609   1000         LD   H,(IX+9)
7075  226B02   1010 L100:   LD   (head-ref),HL
               1020 ;Branch to bypass if head>=tail
7078  2A6B02   1030 loop:   LD   HL,(head-ref)
707B  ED5B6902 1040 L110:   LD   DE,(tail-ref)
707F  A7       1050         AND  A
7080  ED52     1060         SBC  HL,DE
7082  D2C201   1070 L120:   JP   NC,bypass-ref
               1080 ;Initialise highhead and lowtail
               1090 ; IX & IY respectively
7085  DD2A6B02 1100 L130:   LD   IX,(head-ref)
7089  FD2A6902 1110 L140:   LD   IY,(tail-ref)
               1120 ;Calculate pivot string
               1130 ;descriptor address : HL
708D  2A6B02   1140 L150:   LD   HL,(head-ref)


Hisoft GENA3.1 Assembler. Page     3.

7090  ED5B6902 1150 L160:   LD   DE,(tail-ref)
7094  19       1160         ADD  HL,DE
7095  CB3C     1170         SRL  H
7097  CB1D     1180         RR   L
7099  EB       1190         EX   DE,HL
709A  3A7702   1200 L170:   LD   A,(bytes-ref)
709D  47       1210         LD   B,A
709E  210000   1220         LD   HL,0
```

```
70A1  19        1230 mult:   ADD  HL,DE
70A2  10FD      1240         DJNZ mult
70A4  ED5B6D02  1250 L180:   LD   DE,(array-ref)
70A8  19        1260         ADD  HL,DE
                1270 ;Get length and address of
                1280 ;pivot string : plen & ppoint
70A9  ED5B7902  1290 L190:   LD   DE,(offset-ref)
70AD  19        1300         ADD  HL,DE
70AE  7E        1310         LD   A,(HL)
70AF  327002    1320 L200:   LD   (plen-ref),A
70B2  23        1330         INC  HL
70B3  5E        1340         LD   E,(HL)
70B4  23        1350         INC  HL
70B5  56        1360         LD   D,(HL)
70B6  ED537102  1370 L210:   LD   (ppoint-ref),DE
                1380 ;Set pointer to first string
                1390 ;descriptor: sd1point
70BA  210000    1400         LD   HL,0
70BD  ED5B6B02  1410 L220:   LD   DE,(head-ref)
70C1  3A7702    1420 L230:   LD   A,(bytes-ref)
70C4  47        1430         LD   B,A
70C5  19        1440 mult2:  ADD  HL,DE
70C6  10FD      1450         DJNZ mult2
70C8  ED5B6D02  1460 L240:   LD   DE,(array-ref)
70CC  19        1470         ADD  HL,DE
70CD  227302    1480 L250:   LD   (sd1point-ref),HL
                1490 ;Set pointer to second string
                1500 ;descriptor: sd2point
70D0  210000    1510         LD   HL,0
70D3  ED5B6902  1520 L260:   LD   DE,(tail-ref)
70D7  47        1530         LD   B,A
70D8  19        1540 mult3:  ADD  HL,DE
70D9  10FD      1550         DJNZ mult3
70DB  ED5B6D02  1560 L270:   LD   DE,(array-ref)
70DF  19        1570         ADD  HL,DE
70E0  227502    1580 L280:   LD   (sd2point-ref),HL
                1590 ;Get length and address of first
                1600 ;string: C & DE
70E3  2A7302    1610 first:  LD   HL,(sd1point-ref)
70E6  ED5B7902  1620 L290:   LD   DE,(offset-ref)
70EA  19        1630         ADD  HL,DE
70EB  4E        1640         LD   C,(HL)
70EC  23        1650         INC  HL
70ED  5E        1660         LD   E,(HL)
70EE  23        1670         INC  HL
70EF  56        1680         LD   D,(HL)
                1690 ;Compare first string to pivot
                1700 ;branch to proc1 if string<pivot
70F0  0600      1710         LD   B,0
70F2  2A7102    1720 L300:   LD   HL,(ppoint-ref)


Hisoft GENA3.1 Assembler. Page     4.

70F5  1A        1730 comp:   LD   A,(DE)
70F6  BE        1740         CP   (HL)
70F7  3811      1750         JR   C,proc1
```

```
70F9  201E      1760           JR   NZ,second
70FB  04        1770           INC  B
70FC  3A7002    1780 L310:     LD   A,(plen-ref)
70FF  B8        1790           CP   B
7100  2817      1800           JR   Z,second
7102  79        1810           LD   A,C
7103  B8        1820           CP   B
7104  2804      1830           JR   Z,proc1
7106  13        1840           INC  DE
7107  23        1850           INC  HL
7108  20EB      1860           JR   NZ,comp
                1870 ;Add 3 to sd1point
710A  2A7302    1880 proc1:    LD   HL,(sd1point-ref)
710D  ED5B7702  1890 L320:     LD   DE,(bytes-ref)
7111  19        1900           ADD  HL,DE
7112  227302    1910 L330:     LD   (sd1point-ref),HL
                1920 ;increment highhead
7115  DD23      1930           INC  IX
7117  18CA      1940           JR   first
                1950 ;Get length and address of
                1960 ;second string : C & DE
7119  2A7502    1970 second:   LD   HL,(sd2point-ref)
711C  ED5B7902  1980 L340:     LD   DE,(offset-ref)
7120  19        1990           ADD  HL,DE
7121  4E        2000           LD   C,(HL)
7122  23        2010           INC  HL
7123  5E        2020           LD   E,(HL)
7124  23        2030           INC  HL
7125  56        2040           LD   D,(HL)
                2050 ;Compare second string to pivot
                2060 ;branch to proc2 if string>pivot
7126  0600      2070           LD   B,0
7128  EB        2080           EX   DE,HL
7129  ED5B7102  2090 L350:     LD   DE,(ppoint-ref)
712D  1A        2100 comp2:    LD   A,(DE)
712E  BE        2110           CP   (HL)
712F  3811      2120           JR   C,proc2
7131  2020      2130           JR   NZ,over
7133  04        2140           INC  B
7134  79        2150           LD   A,C
7135  B8        2160           CP   B
7136  281B      2170           JR   Z,over
7138  3A7002    2180 L360:     LD   A,(plen-ref)
713B  B8        2190           CP   B
713C  2804      2200           JR   Z,proc2
713E  13        2210           INC  DE
713F  23        2220           INC  HL
7140  20EB      2230           JR   NZ,comp2
                2240 ;Subtract 3 from sd2point
7142  2A7502    2250 proc2:    LD   HL,(sd2point-ref)
7145  ED5B7702  2260 L370:     LD   DE,(bytes-ref)
7149  A7        2270           AND  A
714A  ED52      2280           SBC  HL,DE
714C  227502    2290 L380:     LD   (sd2point-ref),HL
                2300 ;decrement lowtail
```

```
Hisoft GENA3.1 Assembler. Page     5.

714F  FD2B      2310          DEC  IY
7151  18C6      2320          JR   second
                2330 ;Compare sd1point to sd2point
                2340 ;Br. proc3 if sd1point<sd2point
                2350 ;Br. skip if sd1point>sd2point
                2360 ;if = dec lowtail & inc highhead
7153  A7        2370 over:    AND  A
7154  2A7302    2380 L390:    LD   HL,(sd1point-ref)
7157  ED5B7502  2390 L400:    LD   DE,(sd2point-ref)
715B  ED52      2400          SBC  HL,DE
715D  3808      2410          JR   C,proc3
715F  2031      2420          JR   NZ,skip
7161  FD2B      2430          DEC  IY
7163  DD23      2440          INC  IX
7165  182B      2450          JR   skip
                2460 ;swop string descriptors
7167  3A7702    2470 proc3:   LD   A,(bytes-ref)
716A  47        2480          LD   B,A
716B  2A7302    2490 L410:    LD   HL,(sd1point-ref)
716E  1A        2500 swop:    LD   A,(DE)
716F  4E        2510          LD   C,(HL)
7170  EB        2520          EX   DE,HL
7171  71        2530          LD   (HL),C
7172  12        2540          LD   (DE),A
7173  13        2550          INC  DE
7174  23        2560          INC  HL
7175  10F7      2570          DJNZ swop
                2580 ;Add 3 to sd1point
7177  2A7302    2590 L420:    LD   HL,(sd1point-ref)
717A  ED5B7702  2600 L430:    LD   DE,(bytes-ref)
717E  19        2610          ADD  HL,DE
717F  227302    2620 L440:    LD   (sd1point-ref),HL
                2630 ;subtract 3 from sd2point
7182  2A7502    2640 L450:    LD   HL,(sd2point-ref)
7185  A7        2650          AND  A
7186  ED52      2660          SBC  HL,DE
7188  227502    2670 L460:    LD   (sd2point-ref),HL
                2680 ;dec highhead:inc lowtail
718B  DD23      2690          INC  IX
718D  FD2B      2700          DEC  IY
718F  C3DC00    2710 L470:    JP   first-ref
                2720 ;increment stackcounter
7192  216F02    2730 skip:    LD   HL,stackcount-ref
7195  34        2740          INC  (HL)
                2750 ;Calc lowtail-lowhead &
                2760 ; hightail-highead
7196  A7        2770          AND  A
7197  FDE5      2780          PUSH IY
7199  E1        2790          POP  HL
719A  ED5B6B02  2800 L480:    LD   DE,(head-ref)
719E  ED52      2810          SBC  HL,DE
71A0  EB        2820          EX   DE,HL
71A1  A7        2830          AND  A
71A2  2A6902    2840 L490:    LD   HL,(tail-ref)
71A5  DDE5      2850          PUSH IX
```

```
71A7  C1        2860          POP  BC
71A8  ED42      2870          SBC  HL,BC
                2880 ;Compare results & stack larger


Hisoft GENA3.1 Assembler. Page     6.

                2890 ;limits : process smaller limits
71AA  A7        2900          AND  A
71AB  ED52      2910          SBC  HL,DE
71AD  300D      2920          JR   NC,sthigh
71AF  2A6B02    2930 L500:    LD   HL,(head-ref)
71B2  E5        2940          PUSH HL
71B3  FDE5      2950          PUSH IY
71B5  DD226B02  2960 L510:    LD   (head-ref),IX
71B9  C37100    2970 L520:    JP   loop-ref
71BC  DDE5      2980 sthigh:  PUSH IX
71BE  2A6902    2990 L530:    LD   HL,(tail-ref)
71C1  E5        3000          PUSH HL
71C2  FD226902  3010 L540:    LD   (tail-ref),IY
71C6  C37100    3020 L550:    JP   loop-ref
                3030 ;compare stackcounter to zero
71C9  3A6F02    3040 bypass:  LD   A,(stackcount-ref)
71CC  FE00      3050          CP   0
71CE  280F      3060          JR   Z,finish
                3070 ;decrement stackcounter
71D0  3D        3080          DEC  A
71D1  326F02    3090 L560:    LD   (stackcount-ref),A
                3100 ;Pop tail & head from stack
71D4  E1        3110          POP  HL
71D5  226902    3120 L570:    LD   (tail-ref),HL
71D8  E1        3130          POP  HL
71D9  226B02    3140 L580:    LD   (head-ref),HL
71DC  C37100    3150 L590:    JP   loop-ref
71DF  C9        3160 finish:  RET
71E0            3170 Wspace:  DEFS 4
71E4  2000      3180 Ltable:  DEFW end+1-ref
71E6  2900      3190          DEFW Ctable-ref
71E8  7200      3200          DEFW loop+1-ref
71EA  DD00      3210          DEFW first+1-ref
71EC  0401      3220          DEFW proc1+1-ref
71EE  1301      3230          DEFW second+1-ref
71F0  3C01      3240          DEFW proc2+1-ref
71F2  6101      3250          DEFW proc3+1-ref
71F4  8C01      3260          DEFW skip+1-ref
71F6  C301      3270          DEFW bypass+1-ref
71F8  23002C00  3280          DEFW L10+1-ref,L20+1-ref
71FC  39003C00  3290          DEFW L30+1-ref,L40+1-ref
7200  3F004800  3300          DEFW L50+1-ref,L60+1-ref
7204  52005D00  3310          DEFW L70+1-ref,L80+1-ref
7208  66006F00  3320          DEFW L90+1-ref,L100+1-ref
720C  76007C00  3330          DEFW L110+2-ref,L120+1-ref
7210  80008400  3340          DEFW L130+2-ref,L140+2-ref
7214  87008B00  3350          DEFW L150+1-ref,L160+2-ref
7218  94009F00  3360          DEFW L170+1-ref,L180+2-ref
721C  A400A900  3370          DEFW L190+2-ref,L200+1-ref
7220  B100B800  3380          DEFW L210+2-ref,L220+2-ref
```

```
7224  BB00C300  3390        DEFW L230+1-ref,L240+2-ref
7228  C700CE00  3400        DEFW L250+1-ref,L260+2-ref
722C  D600DA00  3410        DEFW L270+2-ref,L280+1-ref
7230  E100EC00  3420        DEFW L290+2-ref,L300+1-ref
7234  F6000801  3430        DEFW L310+1-ref,L320+2-ref
7238  0C011701  3440        DEFW L330+1-ref,L340+2-ref
723C  24013201  3450        DEFW L350+2-ref,L360+1-ref
7240  40014601  3460        DEFW L370+2-ref,L380+1-ref

Hisoft GENA3.1 Assembler. Page     7.

7244  4E015201  3470        DEFW L390+1-ref,L400+2-ref
7248  65017101  3480        DEFW L410+1-ref,L420+1-ref
724C  75017901  3490        DEFW L430+2-ref,L440+1-ref
7250  7C018201  3500        DEFW L450+1-ref,L460+1-ref
7254  89019501  3510        DEFW L470+1-ref,L480+2-ref
7258  9C01A901  3520        DEFW L490+1-ref,L500+1-ref
725C  B001B301  3530        DEFW L510+2-ref,L520+1-ref
7260  B801BD01  3540        DEFW L530+1-ref,L540+2-ref
7264  C001CB01  3550        DEFW L550+1-ref,L560+1-ref
7268  CF01D301  3560        DEFW L570+1-ref,L580+1-ref
726C  D6010000  3570        DEFW L590+1-ref,0

Pass 2 errors: 00

Table used:   1115  from  1200
```

The RSX loader/test program

Program 13.4 is a loader/test program for loading and testing the RSX at any address within the memory pool. The test section sets up a random array of 3 field records. The fields are numbered field zero, field 1 and field 2 respectively and the array is dimensioned DIM (FIELDS%,RECORDS%), where FIELDS% is set to the value 2 (zero is taken as a positive number). You are asked for the number of records, and which field (0 to 2) is to be the subject of the sort. As with the previous test program (Program 13.2) the RSX routine must be called initially with:

CALL <load address>

to log on the RSX and change certain locations for execution at the load address. The RSX is subsequently executed by the following external command:

|SORT2STR,head%,tail%,FIELDS%,FIELDNUM%,@A$(0,0)

The parameters required are:

1 head%, is the minimum indexed record to be included in the sort (usually set to 1).

2 tail%, is the maximum indexed record to be included.
3 FIELDS%, the number of fields in a record minus one or the value which appears in the DIMension command.
4 FIELDNUM%, the index to the field by which the array is to be sorted (0 to n−1), where n is the total number of fields.
5 The @A$(0,0) term passes the base address of the array string descriptors.

Program 13.4 Loader/test program for the rectangular array Quicksort

```
10 REM TEST PROGRAM :
20 REM RECTANGULAR STRING ARRAY SORT (RSX)
40 CLS
50 INPUT"Load address";addr
60 MEMORY ABS(addr)-1
70 LOAD"SORT2STR.BIN",addr
80 CALL addr
90 CLS
100 INPUT"Sort how many 3 field records";RECORDS%
110 FIELDS%=2: REM 3 fields (0,1 & 2)
120 INPUT"Sort which field (0-2)";FIELDNUM%
130 IF FIELDNUM%<0 OR FIELDNUM%>2 THEN 120
140 PRINT
150 REM FILL AND DISPLAY RANDOM ARRAY
160 DIM A$(FIELDS%,RECORDS%)
170 head%=1:tail%=RECORDS%
180 FOR R%=head% TO tail%
190 FOR F%=0 TO FIELDS%
200 B$=""
210 A%=6*RND+1
220 FOR Z%=1 TO A%
230 N%=25*RND
240 K$=CHR$(N%+65)
250 B$=B$+K$
260 NEXT
270 A$(F%,R%)=B$
280 PRINT A$(F%,R%),
290 NEXT
300 NEXT
310 PRINT:PRINT
320 PRINT"SORTING ARRAY"
330 PRINT:PRINT
340 START=TIME/300
350 |SORT2STR,head%,tail%,FIELDS%,FIELDNUM%,@A$(0,
0)
360 T=TIME/300-START
370 FOR R%=head% TO tail%
380 FOR F%=0 TO FIELDS%
390 PRINT A$(F%,R%),
400 NEXT
410 NEXT
420 PRINT
430 PRINT"RECORDS SORTED=";RECORDS%
440 PRINT
```

```
450 PRINT"SORTING TIME=";ROUND(T,2);"SECONDS"
460 PRINT
470 INPUT"Another test (Y/N)";K$
480 K$=UPPER$(K$)
490 IF K$="Y" THEN ERASE A$:GOTO 90 ELSE END
```

Producing a binary file of the RSX

The following step by step guide shows how to produce, load and test the RSX (Program 13.3).

(a) Load the Assembler at, say &2000.
(b) Type in Program 13.3.
(c) Save the source code file by typing

P 10,3570,SORT2STR.RSX

(d) Assemble the code by entering the assembler command A. When asked for table size, respond with 1500.
(e) Save the object code as a binary file by typing the command:

O,,SORT2STR.BIN

(f) Clear memory of the assembler by performing a hard reset then type in Program 13.4. This program will load, run and test the object code file you have just produced. Remember to save the program first! (Murphy's law).

Breakdown of Program 13.3

The listing is similar in many respects to Programs 12.1 and 13.1 so only additional or modified sections need describing. In view of this, readers should refer back to the breakdowns of Programs 12.1 and 13.1 before attempting to decipher the following labels and abbreviations.
Lines 140 to 150: the labels 'bytes' and 'offset' are assigned to the addresses #727E and #7280 respectively.
Lines 700 to 720: initialise the labelled locations 'offset' and 'bytes' to zero.
Lines 810 to 850: pick up the 16 bit value of the BASIC variable, 'FIELDNUM%', from the parameter block. This value is incremented, to take into account array index numbering from zero rather than one. The result is then multiplied by three (the number of bytes in a string descriptor) and stored in the location labelled 'offset'. This value will act as an offset (from the record's first field string descriptor) to the string descriptor associated with a particular key field. Multiplication by three is achieved by a single shift left

(ie×2) then adding the original number.
Lines 880 to 930: pick up the number of fields in a record, again adding 1, then multiplying by 3. The result, stored in the location 'bytes', is thus the total length of the record's string descriptors. These are the number of bytes which may be swapped in position during sorting.
Lines 1140 to 1260: calculate the string descriptor address, associated with the first field of the record containing the current pivot string. (The pivot string descriptor will be offset from this). The calculation is done in three stages. Firstly, 'head' and 'tail' are added together. Secondly, integer division by two is performed by a single shift right. Finally, the result is multiplied by 'bytes' and added to the access table base address 'array'. The formula evaluated is:

string descriptor address=[(head+tail)/2]*bytes

where the integer value is taken within the square brackets.
Lines 1290 to 1360: add 'offset' to the above string descriptor address so as to access the pivot string descriptor associated with the key field. The result, in the HL register pair, is used as a data pointer to pick up the length and address of the key field's pivot string.
Lines 1400 to 1480: calculate the address pointer to the initial string descriptor of the current record, scanning from bottom of list/ sublist, and store the result in 'sd1point'. The equation evaluated is:

sd1point=(head*bytes)+array

Lines 1510 to 1580: calculate the address pointer to the initial string descriptor of the current record, scanning from bottom of list/ sublist, and store the result in 'sd2point'. The equation evaluated is:

sd2point=(tail*bytes)+array

Lines 1610 to 1680: add 'offset' to the string descriptor address associated with the first field of the record containing string1. The result, the address of the key field's string descriptor, is used to pick up the length and address of string1 using implied addressing.
Lines 1880 to 1910: add 'bytes' to the address pointer 'sd1point' so as to point to the next appropriate string descriptor.
Lines 1970 to 2040: add 'offset' to the string descriptor address associated with the first field of the record containing string2. The result, the address of the key field's string descriptor, is used to pick up the length and address of string2 using implied addressing.
Lines 1880 to 1910: subtract 'bytes' from the address pointer 'sd1point' so as to point to the next appropriate string descriptor.
Lines 2470 to 2570: swap the string descriptors, corresponding to the entire record, containing string1 with those of the record containing string2.

Lines 2590 to 2620: add 'bytes' to 'sd1point' so as to point to the next appropriate string descriptor.
Lines 2640 to 2670: subtract 'bytes' from 'sd2point' so as to point to the next appropriate string descriptor.

Summary

1 The Amstrad machines have a dynamic system of memory allocation.
2 An RSX should preferably be location independent or self relocating.
3 The restart RST 6 is guaranteed free for programming by the user.
4 The user restart can be used to extend the Z80 instruction set and can be considered a miniature subroutine.
5 RST 6 is executed by the single byte instruction RST #30.
6 The restart routine must not exceed 8 bytes in the range #0030 to #0037.
7 Self relocating object code can be produced from any subroutine by following four steps.
8 Once loaded, a self relocatable RSX must be called, at its load address, to log its name on to the external command table. At the same time, a short routine converts all 'ref' relative operand addresses to absolute.
9 An RSX can be automatically loaded beneath HIMEM.
10 A data file can be stored in BASIC as a rectangular array or two dimensional array. Fields will occupy one dimension and records the other.
11 When using rectangular arrays always use the zeroth indexed elements or memory will be needlessly wasted.
12 With an array dimensioned

DIM A$(x,y)

the first index (x) is the field index number and the second index (y) is the record number.
13 The term @A$(0,0), returns the string descriptor address of A$(0,0), not the string address itself.
14 The RSX Quicksort of a rectangular array will handle a computer full of records in about one second.

Graphics and direct screen addressing 14

Fast methods of setting up and independently moving multicoloured sprites of any size, without perceptible *flicker*, are explained now. In order to gain maximum speed it is necessary to address screen memory directly. Unfortunately, the rather unusual layout of screen memory and pixel encoding makes this a rather tedious task. However, the principles are not difficult to grasp and can be put to a variety of uses including graphics reinforcement to educational programs and, of course, action games. The example routines presented in this chapter can be incorporated in BASIC/machine code hybrids or as part of a stand-alone machine code programs.

Using resident firmware routines

It is possible, and in many quarters encouraged, to perform graphics operations using *resident firmware* routines. Programs produced in this way, have the advantages of being simple to implement and are guaranteed portable between machines of the same type. A number of calls to various firmware graphics routines may be linked together with a few bytes of connective tissue to perform a full working program. Most firmware routines are, of necessity, general purpose in nature and may perform duties over and above a particular requirement and so waste execution time. The requirement of speed is one of the reasons why programmers resort to the *direct addressing* of screen memory in animated sequences.

Typical action games

Figure 14.1 is a simplified flowchart of an action game, typical of the early eighties. Although much of it is straightforward, problems are often encountered with flicker. The blame rests almost entirely with slow execution speed resulting from the need to continually *draw* and *erase* sprites to simulate movement. It is in this area that we shall now concentrate.

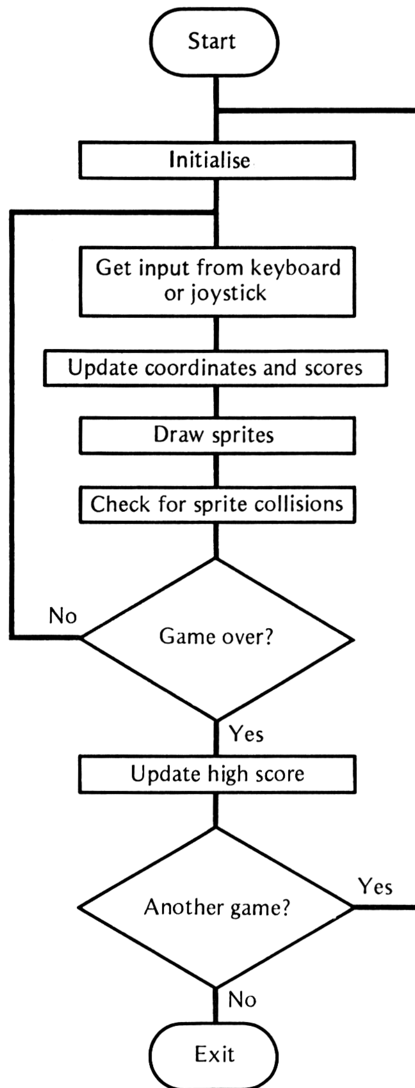


Fig 14.1 Flowchart of a typical action game

Organisation of screen memory

As we have previously remarked, the Amstrad screen memory is set out in an unusual manner. The 16K memory area is, by default, situated at the top of the memory map in the address range #C000 to #FFFF. Each of the 200 pixel lines, regardless of mode, are coded by 80 bytes making 200×80=16000 bytes in all. It follows that

16384(16K)−16000=384 bytes are unused. This region is split up into eight 2K blocks, of which 48 bytes at the end of each block remain unused. The pixel lines, 0,8,16,24,32,40,48...192 are coded sequentially in the first block. The pixel lines, 1,9,17,25,33,41,49,...193 are coded sequentially in the second block and so on. The gaps of eight between the series correspond to the relative spacing of each standard text character row.

In all Modes, the *vertical resolution is fixed at 200 pixels*. The horizontal resolution is all that changes on selecting a different screen mode. In Mode 0, two pixels are encoded by each screen byte, whereas 4 and 8 pixels are encoded by each byte in Modes 1 and 2 respectively. Thus each pixel is bit mapped only in the highest resolution Mode 2. If only one bit is used to encode a pixel it can only be one of two colours coded by a '1' or a '0'. In mode 1, two bits are available enabling four colours. Mode 0, having four bits to encode each pixel, enables a choice of sixteen colours.

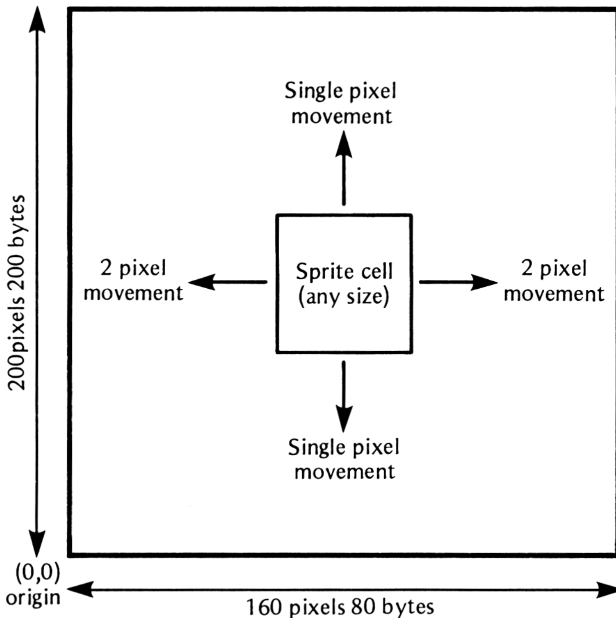


Fig 14.2 Mode zero dynamic graphics resolution (fast method)

Mode Zero

Taking all things into account, the most versatile mode for animated graphics is Mode 0. This mode has a reasonable resolution (160 by 200 pixels), at the same time allowing a maximum of sixteen different pixel colours which can be selected from a choice of 27

inks. A particularly fast graphics method is possible providing we are content with two-pixel horizontal movement at a time. Although not affecting pixel resolution, there is a corresponding reduction in 'dynamic' resolution to 80 by 200. However, this disadvantage is more than compensated for by the ease of manipulation of pixels on the screen. In fact, it simplifies to moving whole bytes around in screen memory. Figure 14.2 (previous page) shows the byte/pixel relationship of screen memory.

How pixels are encoded in a screen memory byte (Mode 0)

For ease of use within the system, the software designers of the Amstrad have again chosen a rather unusual method of encoding pixels within a screen byte. Figure 14.3 shows how the pixel colours must be encoded. The current pen colours in the range 0 to 15 are thus encoded in L_3 L_2 L_1 L_0 (msb to isb) for the left hand pixel and R_3 R_2 R_1 R_0 for the right hand pixel.

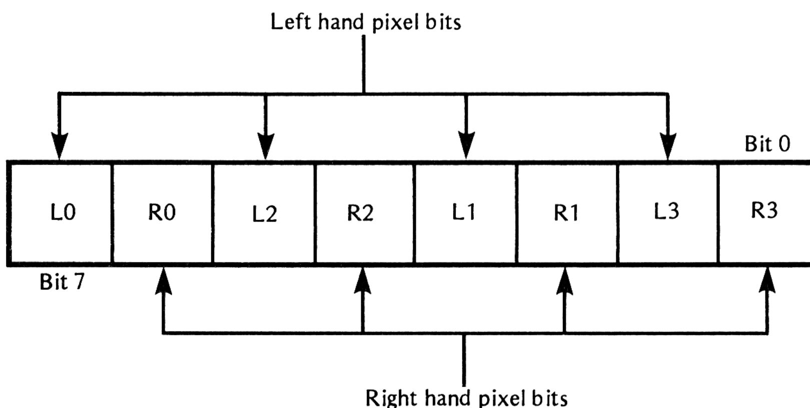


Fig 14.3 How pixels are encoded within screen bytes (Mode 0)

For example, Figure 14.4 shows how to encode a bright red pixel on the left of a bright green pixel in an arbitrary byte of screen memory. With default inks, the code numbers for bright red and bright green are respectively 3 (0011 binary) and 12 (1100 binary).

Shape tables for Mode zero

A sprite, or MOB (Moving OBject), are blanket terms used to decribe any arbitrary shape *together with its associated velocity.* When designing a sprite, it is often convenient to consider it enclosed

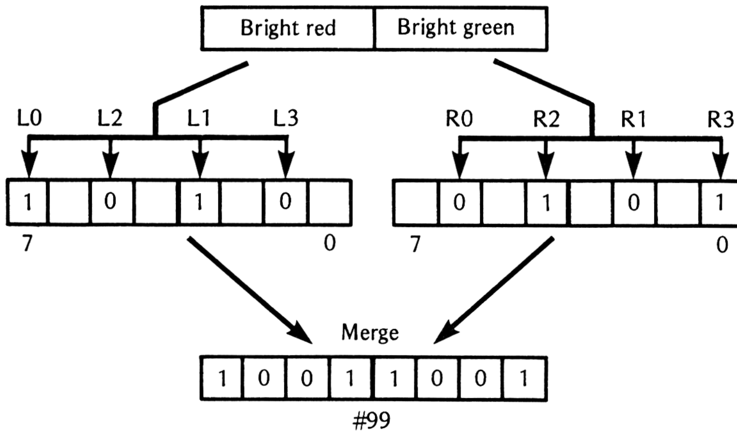


Fig 14.4 An example of pixel encoding (Mode 0)

within a rectangular cell of grid lines where each sub cell represents a pixel. Once the shape is designed, colour or pen numbers can be allotted to each pixel. Any irregular shape can be coded since the background colour can be specified for any unused pixels. It is always a good idea to ensure that a *border of background colour* is present around the shape. By doing this, we need not concern ourselves with erasing any remaining bits and pieces whichever direction the shape is moved. This border would, of course, be two pixels wide on the left and right hand sides of the shape and one pixel high at the top and bottom. Horizontal pairs of pixels are then encoded into bytes by the method described above and finally placed in a data table for direct transfer into screen memory. For convenience, the first two entries in a shape table may be the height and width of the encoded shape in bytes. This data would subsequently be used by a single routine which dumps one or more different shapes into screen memory. Figure 14.5 (overleaf) shows the row by row order in which bytes, corresponding to a 10 by 12 pixel shape rectangle, can be dumped into screen memory. Remember that 5 bytes will contain the data for ten horizontal pixels in Mode zero.

Placing a shape at screen co-ordinates X,Y

Within a program, a shape is often drawn *relative* to a particular pair of X,Y screen co-ordinates. For subsequent ease of sprite manipulation, we can make use of a firmware routine called SCR DOT POSITION (#BC1D), which translates the X,Y co-ordinates to a reference screen address. From then on, direct screen addressing can be used to dump the entire shape table into screen memory (see Figure 14.6). In the example programs of this Chapter, we will be

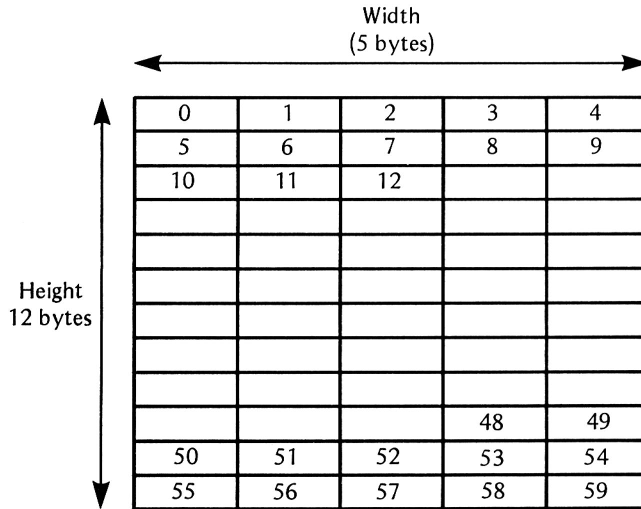


Fig 14.5 Order in which shape table bytes are dumped in screen memory

using the byte corresponding to the top left hand corner of the shape rectangle as this reference address. The details of the firmware routine SCR DOT POSITION (#BC1D) are set out below. For our purposes we can neglect the exit conditions regarding the BC register pair.

SCR DOT POSITION **(BC1D)**

Purpose:
(a) Convert X,Y screen co-ordinates to a screen memory address.
(b) Return a mask byte for chosen pixel.
(c) Return a number one less than the number of pixels in a byte.

Conditions on entry:
The DE and HL register pairs must contain the X and Y co-ordinates respectively of the pixel.

Conditions on exit:
The HL register pair contains the screen memory address. The C register contains the mask byte and the B register contains the number of pixels in a screen byte minus one. Registers AF and DE are corrupted.

Notes:
1 Screen co-ordinates (0,0) refer to the bottom left hand corner of the visible screen.
2 Supplying illegal co-ordinates will produce garbage output.

Addressing screen memory directly

Once an initial screen address is obtained for the chosen co-ordinate position, the shape table bytes are dumped directly into memory.

The unused 48 bytes of each 2K block, mentioned above, need not concern us when addressing screen memory. To obtain the screen memory address of any adjacent pixel pair to the right, simply *add one* to the previous address. To obtain the screen memory address of an adjacent lower pixel pair, *add 2048* (800 hex) to the previous address. However, if the result happens to be greater than #FFFF, subtract 16304 (3FB0 hex). Figure 14.6 shows how a shape can be envisaged in screen memory.

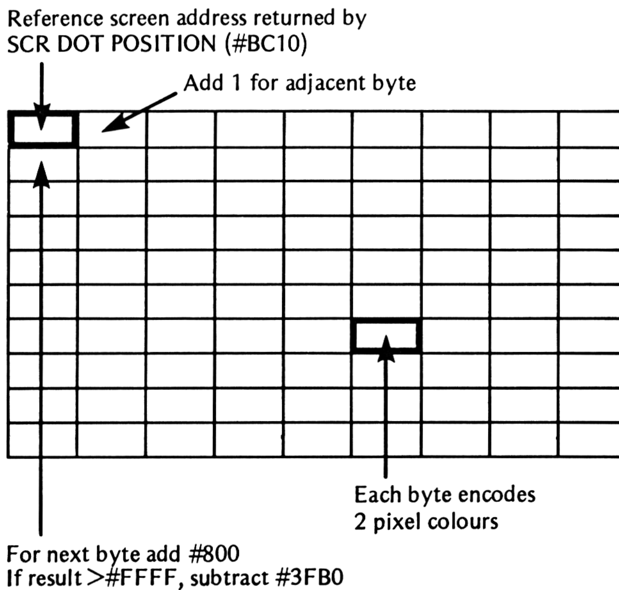


Fig 14.6 A shape rectangle dumped in screen memory

Frame flyback

The frame flyback signal, generated by the 6845 CRT controller chip, is the synchronisation pulse which signals the start of the frame retrace period (moving the CRT tracing spot back to the top left hand corner of the screen). Although the sync pulse itself is rather short, the vertical retrace period is relatively long. For internal reasons, the operating system interrupts this period half way through, resulting in a small proportion of this time being lost to the programmer. During the retrace period, the screen is blanked out to

prevent the retrace lines being visible. Therefore, to simulate the instantaneous appearance or movement of shapes, it is advantageous to employ this period when writing to screen memory. This practice will reduce flicker and other disagreeable effects. There is a firmware routine which holds up execution of programs until the frame flyback pulse is detected. Writing to screen memory should preferably be performed immediately after the firmware call. The routine is:

MC WAIT FLYBACK (#BD19)

There are no entry or exit conditions and all registers and flags are preserved on return.

Changing the mode by a firmware call

When programming in BASIC we change the mode by a simple MODE command. When working in assembly language we need to set up and call the corresponding firmware routine. The routine is called SCR SET MODE and the details are given below.

SCR SET MODE (#BCOE)

Purpose:
To put the screen into a specified mode and ensure that the text and graphics VDUs are set up.

Conditions on entry:
Register A must contain the required mode number in the range 0 to 2.

Conditions on exit:
AF, BC, DE and HL are corrupted and the screen cleared.

Moving a multicoloured shape

Program 14.1 is an example program which sets up and moves a single shape diagonally across the screen. The program is split into three identifiable sections.

(a) Lines 10 to 150 perform initialisation and control.
(b) Lines 180 to 470 arrange for the dumping of the shape table data into screen memory at specified base co-ordinates.
(c) Lines 490 to 760 contain the shape table data.

The shape itself is a simple rectangle of various colours. In practice, of course, the shape table could be modified for displaying anything from a perambulator to a space ship.

Program 14.1 Moving a multicoloured shape

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                       10 ;MOVING A SINGLE
                       20 ;MULTICOLOURED OBJECT
7000                   30          ORG  #7000
7000   97              40          SUB  A
7001   CD0EBC          50          CALL #BC0E
7004   1E00            60          LD   E,0
7006   2E0A            70          LD   L,10
7008   0646            80          LD   B,70
700A   1C              90 loop:    INC  E
700B   1C             100          INC  E
700C   2C             110          INC  L
700D   CD19BD         120          CALL #BD19
7010   CD1670         130          CALL draw
7013   10F5           140          DJNZ loop
7015   C9             150          RET
                      160 ;
                      170 ;
7016   C5             180 draw:    PUSH BC
7017   D5             190          PUSH DE
7018   E5             200          PUSH HL
7019   2600           210          LD   H,0
701B   1600           220          LD   D,0
701D   CD1DBC         230          CALL #BC1D
7020   ED4B4570       240          LD   BC,(shape)
7024   114770         250          LD   DE,shape+2
7027   C5             260 vert:    PUSH BC
7028   E5             270          PUSH HL
7029   1A             280 horiz:   LD   A,(DE)
702A   77             290          LD   (HL),A
702B   23             300          INC  HL
702C   13             310          INC  DE
702D   10FA           320          DJNZ horiz
702F   E1             330          POP  HL
7030   010008         340          LD   BC,#800
7033   09             350          ADD  HL,BC
7034   3006           360          JR   NC,over
7036   01B03F         370          LD   BC,#3FB0
7039   97             380          SUB  A
703A   ED42           390          SBC  HL,BC
703C   C1             400 over:    POP  BC
703D   0D             410          DEC  C
703E   B9             420          CP   C
703F   20E6           430          JR   NZ,vert
7041   E1             440          POP  HL
7042   D1             450          POP  DE
7043   C1             460          POP  BC
7044   C9             470          RET
                      480
7045   090A           490 shape:   DEFB 9,10
7047   00000000       500          DEFB #00,#00,#00,#00
704B   00000000       510          DEFB #00,#00,#00,#00
```

```
704F  0000        520          DEFB #00,#00
7051  00999999    530          DEFB #00,#99,#99,#99
7055  99999999    540          DEFB #99,#99,#99,#99
7059  9900        550          DEFB #99,#00
705B  00666666    560          DEFB #00,#66,#66,#66

Hisoft GENA3.1 Assembler. Page     2.

705F  66666666    570          DEFB #66,#66,#66,#66
7063  6600        580          DEFB #66,#00
7065  009999F8    590          DEFB #00,#99,#99,#F8
7069  F8F8F899    600          DEFB #F8,#F8,#F8,#99
706D  9900        610          DEFB #99,#00
706F  006666F4    620          DEFB #00,#66,#66,#F4
7073  F4F4F466    630          DEFB #F4,#F4,#F4,#66
7077  6600        640          DEFB #66,#00
7079  009999F8    650          DEFB #00,#99,#99,#F8
707D  F8F8F899    660          DEFB #F8,#F8,#F8,#99
7081  9900        670          DEFB #99,#00
7083  00666666    680          DEFB #00,#66,#66,#66
7087  66666666    690          DEFB #66,#66,#66,#66
708B  6600        700          DEFB #66,#00
708D  00999999    710          DEFB #00,#99,#99,#99
7091  99999999    720          DEFB #99,#99,#99,#99
7095  9900        730          DEFB #99,#00
7097  00000000    740          DEFB #00,#00,#00,#00
709B  00000000    750          DEFB #00,#00,#00,#00
709F  0000        760          DEFB #00,#00

Pass 2 errors: 00

Table used:     81   from   237
```

Breakdown of Program 14.1

Line 30: instructs assembly at address #7000.
Lines 40 to 50: set up Mode 0 by setting the A register to zero followed by a call to SCR SET MODE.
Lines 60 to 70: set the initial X,Y screen co-ordinates where the shape is to be placed on the screen. The E register holds the X co-ordinate and the L register the Y co-ordinate. The screen co-ordinates can never exceed 160 by 200 so they can always be accommodated within a single register.
Line 80: initialises the loop counter to 70 decimal. The loop itself will cause the shape to traverse 2×70 pixel positions across the screen and 70 pixel positions upward.
Lines 90 to 100: increment the X co-ordinate twice since two-pixel movement must be used horizontally with the method adopted.
Line 110: increments the Y co-ordinate once. (Single pixel movement is allowed vertically).
Line 120: a firmware call to MC WAIT FLYBACK holds up program

execution until the frame flyback pulse is received.
Line 130: calls the subroutine 'draw' which dumps the shape table data into screen memory at the specified X,Y co-ordinates.
Lines 140 to 150: check for loop termination and returns control to the calling program.
Lines 180 to 200: push the register pairs BC, DE and HL on the stack to preserve essential contents. (The contents are restored before return).
Lines 210 to 230: call the firmware routine SCR DOT POSITION (#BC1D). The X co-ordinate is expected in the DE register pair and the Y co-ordinate in the HL register pair. The maximum possible co-ordinates in Mode 0 are 160 for X and 200 for Y so they both lie within the compass of a single register. Therefore, the D and H registers, which normally hold the high bytes in other modes, can be cleared to zero. On return the HL register pair will hold the corresponding screen address.
Line 240: loads the first two data bytes of the shape table, corresponding to the height and width in bytes, into the BC register pair. The C register holds the height and the B register holds the width, in this case, 9 and 10 respectively.
Line 250: uses the DE register as a data pointer and is loaded with the address of the first shape table byte. Notice that the assembler is instructed to add 2 to the address label 'shape' so as to skip the width and height bytes above.
Lines 260 to 270: stack the values held in the BC and HL register pairs for later use.
Lines 280 to 290: use implied addressing to pick up a single data byte from the shape table and dump it into screen memory.
Lines 300 to 310: increment the address pointers to the data table and screen memory. These are DE and HL respectively.
Line 320: terminates the loop once a row of shape table bytes have been dumped into screen memory. This occurs when the B register, containing the width value, has decremented to zero.
Line 330: restores the screen address value stacked in line 270 into the HL register pair.
Lines 340 to 350: add the hex value #800 to the HL register pair to obtain the screen memory address of the pixel pair immediately below the previous location stacked in line 270. This denotes the start address of the next row of shape table bytes to be transferred to screen memory.
Line 360: checks if the previous result is greater than #FFFF. If it is, the carry will be set since #FFFF is the highest number that can be stored in a register pair. If the carry is clear a branch is made to 'over'.
Lines 370 to 390: subtract the necessary #3FB0 from the contents of the HL register pair if the carry happens to be set. SUB A is used in line 380 for two reasons. Firstly, it clears the carry flag for the SBC instruction and secondly, it clears the A register for a later comparison instruction in line 420.

Line 400: restores the width and current height index to the BC register pair.
Lines 420 to 430: decrement the current height index and compare it to register A, previously set to zero. If Register C is non zero, a branch is made back to 'vert' to trace out another row of shape table bytes.
Lines 440 to 470: restore the values of the BC, DE and HL register pairs to those present when the subroutine 'draw' was called.
Lines 490 to 760: contain the shape table data. The first two bytes in line 490 are the height and width of the table in bytes. The actual table, lines 500 to 760, correspond to 9 rows of 10 bytes. In order to display all the data in the object code assembler field, it was necessary to restrict each DEFB directive to a maximum of four byte expressions. Therefore each row of screen memory data bytes are coded in three sequential line numbers.

Moving multicoloured objects independently

Program 14.2 Moving three multicoloured shapes

```
Hisoft GENA3.1 Assembler. Page     1.

Pass 1 errors: 00

                       10 ;MOVING MULTICOLOURED
                       20 ;OBJECTS INDEPENDENTLY
7500                   30 table:  EQU  #7500
7502                   40 coord:  EQU  #7502
7000                   50         ORG  #7000
7000   CD1170          60         CALL init
7003   0646            70         LD   B,70
7005   CD2870          80 repeat: CALL update
7008   CD19BD          90         CALL #BD19
700B   CD3770         100         CALL screen
700E   10F5           110         DJNZ repeat
7010   C9             120         RET
                      130 ;
7011   97             140 init:   SUB  A
7012   CD0EBC         150         CALL #BC0E
7015   21000A         160         LD   HL,#0A00
7018   220275         170         LD   (coord),HL
701B   218C64         180         LD   HL,#648C
701E   220475         190         LD   (coord+2),HL
7021   21648C         200         LD   HL,#8C64
7024   220675         210         LD   (coord+4),HL
7027   C9             220         RET
                      230 ;
7028   210275         240 update: LD   HL,coord
702B   34             250         INC  (HL)
702C   34             260         INC  (HL)
702D   23             270         INC  HL
702E   34             280         INC  (HL)
702F   23             290         INC  HL
7030   35             300         DEC  (HL)
```

```
7031  35        310          DEC   (HL)
7032  23        320          INC   HL
7033  23        330          INC   HL
7034  23        340          INC   HL
7035  35        350          DEC   (HL)
7036  C9        360          RET
                370 ;
7037  C5        380 screen:  PUSH  BC
7038  D5        390          PUSH  DE
7039  E5        400          PUSH  HL
703A  219570    410          LD    HL,shape1
703D  220075    420          LD    (table),HL
7040  ED5B0275  430          LD    DE,(coord)
7044  6A        440          LD    L,D
7045  CD6870    450          CALL  draw
7048  21F170    460          LD    HL,shape2
704B  220075    470          LD    (table),HL
704E  ED5B0475  480          LD    DE,(coord+2)
7052  6A        490          LD    L,D
7053  CD6870    500          CALL  draw
7056  212771    510          LD    HL,shape3
7059  220075    520          LD    (table),HL
705C  ED5B0675  530          LD    DE,(coord+4)
7060  6A        540          LD    L,D
7061  CD6870    550          CALL  draw
7064  E1        560          POP   HL


Hisoft GENA3.1 Assembler. Page     2.

7065  D1        570          POP   DE
7066  C1        580          POP   BC
7067  C9        590          RET
                600 ;
                610 ;
7068  C5        620 draw:    PUSH  BC
7069  2600      630          LD    H,0
706B  1600      640          LD    D,0
706D  CD1DBC    650          CALL  #BC1D
7070  EB        660          EX    DE,HL
7071  2A0075    670          LD    HL,(table)
7074  4E        680          LD    C,(HL)
7075  23        690          INC   HL
7076  46        700          LD    B,(HL)
7077  23        710          INC   HL
7078  EB        720          EX    DE,HL
7079  C5        730 vert:    PUSH  BC
707A  E5        740          PUSH  HL
707B  1A        750 horiz:   LD    A,(DE)
707C  77        760          LD    (HL),A
707D  23        770          INC   HL
707E  13        780          INC   DE
707F  10FA      790          DJNZ  horiz
7081  E1        800          POP   HL
7082  010008    810          LD    BC,#800
7085  09        820          ADD   HL,BC
7086  3006      830          JR    NC,over
```

```
7088  01B03F     840          LD   BC,#3FB0
708B  97         850          SUB  A
708C  ED42       860          SBC  HL,BC
708E  C1         870 over:    POP  BC
708F  0D         880          DEC  C
7090  B9         890          CP   C
7091  20E6       900          JR   NZ,vert
7093  C1         910          POP  BC
7094  C9         920          RET
                 930
7095  090A       940 shape1:  DEFB 9,10
7097  00000000   950          DEFB #00,#00,#00,#00
709B  00000000   960          DEFB #00,#00,#00,#00
709F  0000       970          DEFB #00,#00
70A1  00999999   980          DEFB #00,#99,#99,#99
70A5  99999999   990          DEFB #99,#99,#99,#99
70A9  9900      1000          DEFB #99,#00
70AB  00666666  1010          DEFB #00,#66,#66,#66
70AF  66666666  1020          DEFB #66,#66,#66,#66
70B3  6600      1030          DEFB #66,#00
70B5  009999F8  1040          DEFB #00,#99,#99,#F8
70B9  F8F8F899  1050          DEFB #F8,#F8,#F8,#99
70BD  9900      1060          DEFB #99,#00
70BF  006666F4  1070          DEFB #00,#66,#66,#F4
70C3  F4F4F466  1080          DEFB #F4,#F4,#F4,#66
70C7  6600      1090          DEFB #66,#00
70C9  009999F8  1100          DEFB #00,#99,#99,#F8
70CD  F8F8F899  1110          DEFB #F8,#F8,#F8,#99
70D1  9900      1120          DEFB #99,#00
70D3  00666666  1130          DEFB #00,#66,#66,#66
70D7  66666666  1140          DEFB #66,#66,#66,#66


Hisoft GENA3.1 Assembler. Page     3.

70DB  6600      1150          DEFB #66,#00
70DD  00999999  1160          DEFB #00,#99,#99,#99
70E1  99999999  1170          DEFB #99,#99,#99,#99
70E5  9900      1180          DEFB #99,#00
70E7  00000000  1190          DEFB #00,#00,#00,#00
70EB  00000000  1200          DEFB #00,#00,#00,#00
70EF  0000      1210          DEFB #00,#00
70F1  0C04      1220 shape2:  DEFB 12,4
70F3  00000000  1230          DEFB #00,#00,#00,#00
70F7  00666600  1240          DEFB #00,#66,#66,#00
70FB  00999900  1250          DEFB #00,#99,#99,#00
70FF  00666600  1260          DEFB #00,#66,#66,#00
7103  00999900  1270          DEFB #00,#99,#99,#00
7107  00666600  1280          DEFB #00,#66,#66,#00
710B  00999900  1290          DEFB #00,#99,#99,#00
710F  00F8F800  1300          DEFB #00,#F8,#F8,#00
7113  00F4F400  1310          DEFB #00,#F4,#F4,#00
7117  00F8F800  1320          DEFB #00,#F8,#F8,#00
711B  00F4F400  1330          DEFB #00,#F4,#F4,#00
711F  00F8F800  1340          DEFB #00,#F8,#F8,#00
7123  00000000  1350          DEFB #00,#00,#00,#00
7127  0704      1360 shape3:  DEFB 7,4
7129  00000000  1370          DEFB #00,#00,#00,#00
```

```
712D  00666600  1380            DEFB #00,#66,#66,#00
7131  00999900  1390            DEFB #00,#99,#99,#00
7135  00666600  1400            DEFB #00,#66,#66,#00
7139  00999900  1410            DEFB #00,#99,#99,#00
713D  00666600  1420            DEFB #00,#66,#66,#00
7141  00000000  1430            DEFB #00,#00,#00,#00

Pass 2 errors: 00

Table used:    171  from    371
```

Program 14.2 is an example of how to move three shapes all following independent trajectories. As far as the eye is concerned, movement takes place simultaneously, smoothly and with no perceptible flicker. The routine is split into six different sections.

1 A main control loop (lines 60 to 120).
2 An initialisation routine, (lines 140 to 220) which sets Mode 0 and the start-up screen co-ordinates for the three shapes.
3 An update section which adjusts the co-ordinates to new values. In the example we have simply incremented or decremented them to show how movement is effected.
4 A screen section which draws the three shapes according to the co-ordinates supplied from the update section. For each shape to be drawn a call is made to the 'draw' routine.
5 A routine for drawing a shape depending on which shape table address and X,Y co-ordinates are passed over to it from the screen section.
6 A group of three separate shape data tables.

Co-ordinate blocks

A section of memory can be used to store the updated co-ordinates corresponding to the screen positions of various shapes. Since only one byte is needed to store each X and Y co-ordinate in Mode 0, we can set up a sequential block of locations storing the X and Y co-ordinates of each shape. In Program 14.2, the area of memory reserved for this six byte co-ordinate block is #7500 and is shown diagramatically in Figure 14.7. These locations are accessed sequentially using implied addressing and set up as required prior to displaying an updated screen.

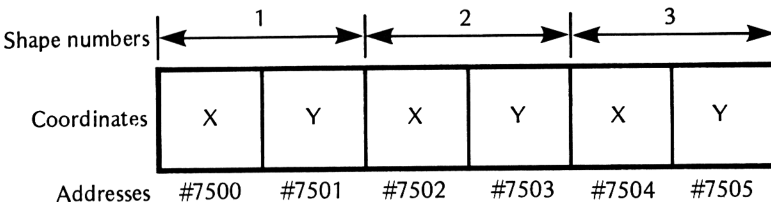


Fig 14.7 Co-ordinate block for Mode 0

The draw routine, as it stands in Program 14.1, needs minor modification before it can be used for multiple shapes. The operands of the instructions in lines 240 and 250 will always be the address of a *single* shape data table. There are two methods of modification to allow other shape table addresses to be incorporated at this point.

1 Change the operand bytes within the subroutine itself by the method described in Chapter 13. However, it must be pointed out that this practice is frowned upon in academic circles.
2 Alter the draw subroutine so that it expects its current shape table address to be present in some common fixed location. This location can be loaded with the appropriate address prior to the subroutine call. We have chosen this approach in Program 14.2.

The example, Program 14.2 moves the first shape *diagonally* across the screen, the second shape *horizontally* and the third *vertically*. The multicoloured shapes, coded in the data tables, have been deliberately kept simple since the actual shape is immaterial at this stage. However, it should show the kind of effects possible using this fast, Mode 0 graphics, method.

Breakdown of Program 14.2

Line 30: assigns the label 'table' to the address used to store the current shape table address.
Line 40: assigns the label 'coord' to the address of the co-ordinate block.
Line 50: forces assembly at #7000.
Line 60: calls the initialisation routine (line 140).
Lines 70 to 110: form the main program loop which controls the sequence of subroutine calls. The loop repeats 70 times in this simple example program. The call in line 90 is an MC FRAME FLYBACK call.
Lines 140 to 150: set up the Mode 0 screen.
Lines 160 to 210: load the co-ordinate block with the initial values of the 3 shape co-ordinates. The assembler expression evaluator is used to add 2 to the previous address of 'coord' each time to store the data at the correct co-ordinate block addresses. With each HL register pair load, the X co-ordinate data must be in the L register and the Y co-ordinate data in the H register.
Lines 240 to 360: update the co-ordinates of the shapes in the co-ordinate block (see Figure 14.7). The order in which this is done is shape1, shape2, shape3. Implied addressing is used to update these locations. Remember that 2 pixel movement (increment X co-ordinates twice) is required for horizontal movement.
Lines 380 to 400: push the register pairs BC, DE and HL, which hold important information, on the stack. (These registers will be restored before return).
Lines 410 to 420: store the address of the shape1 table in the location labelled 'table'.

Lines 430 to 450: the X,Y co-ordinates for the first shape are loaded into the DE register pair. The X co-ordinate in E and the Y co-ordinate in D. The D register contents are then copied into the L register in line 440. This sets up the entry requirements to SCR DOT POSITION (#BC1D) in the draw subroutine, called at line 450. (The H and D registers are set to zero in Mode 0).
Lines 460 to 500: repeat the above for shape2.
Lines 510 to 550: repeat the above process for shape 3.
Lines 560: to 580: restore the register values to that present on entry to the subroutine.
Line 620: saves the B register, which contains vital data, on the stack.
Lines 630 to 650: set up and call the firmware routine SCR DOT POSITION (#BC1D). The X co-ordinate is expected in the DE register pair and the Y co-ordinate in the HL register pair. The maximum possible co-ordinates in Mode 0 are 160 for X and 200 for Y so they both lie within the compass of a single register. Therefore, the D and H registers, which normally hold the high bytes, can be cleared to zero. On return the HL register pair will hold the corresponding screen address.
Line 660: the screen address in HL is temporarily moved to the DE register pair. Thus freeing the HL register for implied addressing.
Line 670: the HL register pair is loaded with the current shape table address and thus acts as the data table pointer.
Lines 680 to 710: load by implied addressing the first two data bytes of the current shape table, corresponding to its height and width in bytes, into the BC register pair. The C register holds the height and the B register holds the width.
Line 720: performs a double function with the one instruction. Firstly, the HL register pair, which is the current shape table data pointer, is transferred to the DE register pair. At the same time, the current screen memory address previously transferred to the DE register pair is restored to the HL register pair.
Lines 730 to 920: have identical coding to that found in lines 260 to 470 of Program 14.1.
Lines 940 to 1210: contain the shape1 table data. The first two bytes in line 940 are the height and width of the shape table in bytes. The actual table, lines 950 to 1210, correspond to 9 rows of 10 bytes. In order to display all the data in the object code assembler field, it was necessary to restrict each DEFB directive to a maximum of four-byte expressions. Therefore each row of screen memory data bytes are coded in three sequential line numbers.
Lines 1220 to 1350: contain the shape2 data.
Lines 1360 to 1430: contain the shape3 data.

Shape tables

The coding of shape table data by hand can be tedious. If this style of graphics manipulation is found interesting, much time can be

saved, in the long run, by designing a program that produces shape table data as output. The program could be written mainly in BASIC employing one or two of the routines in this chapter to display the resulting shape. Programs of this type usually display a large rectangular cell to specified dimensions. This is then subdivided by *grid lines* into sub cells equivalent to a *single pixel.* The user then moves the cursor round the sub cells specifying which pixel colour is required. From this data (which can be conveniently stored in a rectangular array) the various shape data bytes can be encoded and displayed on the screen. This method of producing shape tables is particularly useful for producing an animated sequence where each shape is varied in outline as it is moved.

Summary

1 Using the graphics firmware is recommended but direct screen addressing works faster.
2 Moving sprites by alternately drawing and erasing is the main cause of flicker and jerkiness.
3 Enclosing a sprite in a border composed of the background colour eliminates the need to erase.
4 The first two entries in a shape table can be the height and width of the encoded shape.
5 To obtain address of next pixel pair to the right, add 1.
6 To obtain address of next pixel below, add 2048 (800 hex).
7 It is best to write to screen memory immediately after a call to MC wait flyback.

Appendices

APPENDIX 1: THE Z80A INSTRUCTION SET

Note: x=status flag updated

Abbreviations used in Op-code:

```
r      IX or IY (0=IX,1=IY)
rr     register pair code:
       BC=00,DE=01,HL=10,SP=11 for rp,AF=11 for pr.
rrr    single register code:
       B=000,C=001,D=010,E=011,H=100,L=101,A=111
bbb    bit position, 000 to 111: b=bit 0 to 7
ccc    condition code:
       000=non zero (NZ)
       001=zero (Z)
       010=no carry (NC)
       011=carry (C)
       100=parity odd (PO) P=0
       101=parity even (PE) P=1
       110=positive (P)
       111=negative (M)
sss    restart code (000 to 111)
```

Abbreviations used in source code:

```
reg    A,B,C,D,E,H, or L
xy     IX or IY: d=displacement byte constant.
rp     register pair BC,DE,HL or SP
pr     register pair BC,DE,HL or AF
pp     BC,DE,IX or SP
qq     BC,DE,IY or SP
cond   condition code
```

JUMP INSTRUCTIONS

Source code	Op-code	C	Z	S	P/O	Ac	N	Bt	Ck
JP cond,label	11ccc010							3	10
JR C,label	38							2	7/12
JR NC,label	30							2	7/12
JR Z,label	28							2	7/12
JR NZ,label	20							2	7/12
DJNZ,label	10							2	8/13
JP label	C3							3	10
JR label	18							2	12
JP (HL)	E9							1	4
JP (IX)	DD E9							2	8
JP (IY)	FD E9							2	8

COMPARISON INSTRUCTIONS

Source code	Op-code	C	Z	S	P/O	Ac	N	Bt	Ck
CP (HL)	BE	x	x	x	0	x	1	1	7
CP (xy+d)	11r11101 BE	x	x	x	0	x	1	3	19
CP data	FE	x	x	x	0	x	1	2	7
CP reg	10111rrr	x	x	x	0	x	1	1	4

LOAD INSTRUCTIONS:

Source code	Op-code	C	Z	S	P/O	Ac	N	Bt	Ck
LD A,(addr)	3A							3	13
LD HL,(addr)	2A							3	16
LD rp,(addr)	ED 01rr1011							4	20
LD xy,(addr)	11r11101 2A							4	20
LD (addr),A	32							3	13
LD (addr),HL	22							3	16
LD (addr),rp	ED 01rr0011							4	20
LD (addr),xy	11r11101 22							4	20
LD A,(BC)	0A							1	7
LD A,(DE)	1A							1	7
LD reg,(HL)	01rrr110							1	7
LD (BC),A	02							1	7
LD (DE),A	12							1	7
LD (HL),reg	01110rrr							1	7
LD reg,(xy+d)	11r11101 01rrr110							3	19
LD (xy+d),reg	11r11101 01110rrr							3	19
LD reg,data	00rrr110							2	7
LD rp,data	00rr0001							3	10
LD xy,data	11r11101 21							4	14
LD (HL),data	36							2	10
LD (xy+d)	11r11101 36							4	19
LD dst,src	01rrrrrr							1	4
LD A,I	ED 57		x	x	1	0	0	2	9
LD A,R	ED 5F		x	x	1	0	0	2	9
LD I,A	ED 47							2	9
LD R,A	ED 4F							2	9
LD SP,HL	F9							1	6
LD SP,xy	11r11101 F9							2	10
EX DE,HL	EB							1	4
EX AF,AF'	08							1	4
EXX	D9							1	4

BLOCK TRANSFER AND SEARCH INSTRUCTIONS:

Source code	Op-code	C	Z	S	P/O	Ac	N	Bt	Ck
LDIR	ED B0				0	0	0	2	21/16
LDDR	ED B8				0	0	0	2	21/16
LDI	ED A0				x	0	0	2	16
LDD	ED A8				x	0	0	2	16
CPDR	ED B9		x	x	x	x	1	2	20/16
CPIR	ED B1		x	x	x	x	1	2	20/16
CPI	ED A1		x	x	x	x	1	2	16
CPD	ED A9		x	x	x	x	1	2	16

ARITHMETIC INSTRUCTIONS:

Source code	Op-code	C	Z	S	P/O	Ac	N	Bt	Ck
ADD A,(HL)	86	x	x	x	O	x	O	1	7
ADD A,(xy+d)	11r11101 86	x	x	x	O	x	O	3	19
ADC A,(HL)	8E	x	x	x	O	x	O	1	7
ADC A,(xy+d)	11r11101 8E	x	x	x	O	x	O	3	19
SUB (HL)	96	x	x	x	O	x	1	1	7
SUB (xy+d)	11r11101 96	x	x	x	O	x	1	3	19
SBC A,(HL)	9E	x	x	x	O	x	1	1	7
SBC A,(xy+d)	11r11101 9E	x	x	x	O	x	1	3	19
ADD A,data	C6	x	x	x	O	x	O	2	7
ADC A,data	CE	x	x	x	O	x	O	2	7
SUB data	D6	x	x	x	O	x	1	2	7
SBC A,data	DE	x	x	x	O	x	1	2	7
ADD A,reg	10000rrr	x	x	x	O	x	O	1	4
ADC A,reg	10001rrr	x	x	x	O	x	O	1	4
SUB reg	10010rrr	x	x	x	O	x	1	1	4
SBC A,reg	10011rrr	x	x	x	O	x	1	1	4
ADD HL,rp	00rr1001	x				?	O	1	11
ADC HL,rp	ED 01rr1010	x	x	x	O	?	O	2	15
SBC HL,rp	ED 01rr0010	x	x	x	O	?	1	2	15
ADD IX,pp	DD 00rr1001	x				?	O	2	15
ADD IY,qq	FD 00rr1001	x				?	O	2	15
DAA	27	x	x	x	P	x		1	4
INC (HL)	34		x	x	O	x	O	1	11
INC (xy+d)	11r11101 34		x	x	O	x	O	3	23
DEC (HL)	35		x	x	O	x	1	1	11
DEC (xy+d)	11r11101 35		x	x	O	x	1	3	23
INC reg	00rrr100		x	x	O	x	O	1	4
INC rp	00rr0011							1	6
INC xy	11r11101 23							2	10
DEC reg	00rrr101		x	x	O	x	1	1	4
DEC rp	00rr1011							1	6
DEC xy	11r11101 2B							2	10
CPL	2F					1	1	1	4
NEG	ED 44	x	x	x	O	x	1	2	8

STACK INSTRUCTIONS:

Source code	Op-code	C	Z	S	P/O	Ac	N	Bt	Ck
PUSH pr	11rr0101							1	11
PUSH IX	11011101 E5							2	15
PUSH IY	11111101 E5							2	15
POP pr	11rr0001							1	10
POP IX	11011101 E1							2	14
POP IY	11111101 E1							2	14
EX (SP),HL	E3							1	19
EX (SP),IX	11011101 E3							2	23
EX (SP),IY	11111101 E3							2	23

LOGICAL INSTRUCTIONS

Source code	Op-code	C	Z	S	P/O	Ac	N	Bt	Ck
AND (HL)	A6	0	x	x	p	1	0	1	7
AND (xy+d)	11r11101 A6	0	x	x	P	1	0	3	19
OR (HL)	B6	0	x	x	P	1	0	1	7
OR (xy+d)	11r11101 B6	0	x	x	P	1	0	3	19
XOR (HL)	AE	0	x	x	P	1	0	1	7
XOR (xy+d)	11r11101 AE	0	x	x	P	1	0	3	19
AND data	E6	0	x	x	P	1	0	2	7
OR data	F6	0	x	x	P	1	0	2	7
XOR data	EE	0	x	x	P	1	0	2	7
AND reg	10000rrr	0	x	x	P	1	0	1	4
OR reg	10110rrr	0	x	x	P	1	0	1	4
XOR reg	10101rrr	0	x	x	P	1	0	1	4

BIT MANIPULATIONS

Source code	Op-code	C	Z	S	P/O	Ac	N	Bt	Ck
BIT b,reg	CB 01bbbrrr		x	?	?	1	0	2	8
BIT b,(HL)	CB 01bbb110		x	?	?	1	0	2	12
BIT b,(xy+d)	11r11101 CB		x	?	?	1	0	4	20
SET b,reg	CB 11bbbrrr							2	8
SET b,(HL)	CB 11bbb110							2	15
SET b,(xy+d)	11r11101 CB							4	23
	d 11bbb110								
RES b,reg	CB 10bbbrrr							2	8
RES b,(HL)	CB 10bbb110							2	15
RES b,(xy+d)	11x11101 CB							4	23
	d 10bbb110								

SHIFT AND ROTATE INSTRUCTIONS

Source code	Op-code	C	Z	S	P/O	Ac	N	Bt	Ck
RLC (HL)	CB 06	x	x	x	P	0	0	2	15
RLC (xy+d)	11r11101 CB	x	x	x	P	0	0	4	23
	d 06								
RL (HL)	CB 16	x	x	x	P	0	0	2	15
RL (xy+d)	11r11101 CB	x	x	x	P	0	0	4	23
	d 16								
RRC (HL)	CB 0E	x	x	x	P	0	0	2	15
RCC (xy+d)	11r11101 CB	x	x	x	P	0	0	4	23
	d 0E								
RR (HL)	CB 1E	x	x	x	P	0	0	2	15
RR (xy+D)	11r11101 CB	x	x	x	P	0	0	4	23
	d 1E								
SLA (HL)	CB 26	x	x	x	P	0	0	2	15
SLA (xy+d)	11r11101 CB	x	x	x	P	0	0	4	23
	d 26								
SRA (HL)	CB 2E	x	x	x	P	0	0	2	15
SRA (xy+d)	11r11101 CB	x	x	x	p	0	0	4	23
	d 2E								
SRL (HL)	CB 3E	x	x	x	P	0	0	2	15
SRL (xy+d)	11r11101 CB	x	x	x	P	0	0	4	23
	d 3E								
RLCA	07	x				0	0	1	4
RLA	17	x				0	0	1	4
RRCA	0F	x				0	0	1	4
RRA	1F	x				0	0	1	4
RLC reg	CB 00000rrr	x	x	x	P	0	0	2	8
RL reg	CB 00010rrr	x	x	x	p	0	0	2	8
RRC reg	CB 00001rrr	x	x	x	P	0	0	2	8
RR reg	CB 00011rrr	x	x	x	P	0	0	2	8
SLA reg	CB 00100rrr	x	x	x	P	0	0	2	8
SRA reg	CB 00101rrr	x	x	x	P	0	0	2	8
SRL reg	CB 00111rrr	x	x	x	P	0	o	2	8
RLD	ED 6F		x	x	P	0	0	2	18
RRD	ED 67		x	x	P	0	0	2	18

INPUT/OUTPUT INSTRUCTIONS:

Source code	Op-code	C	Z	S	P/O	Ac	N	Bt	Ck
IN A,(port)	DB							2	10
IN reg,(C)	ED 01rrr000		x	x	P	x	0	2	11
INIR	ED B2		1	?	?	?	1	2	21/16
INDR	ED BA		1	?	?	?	1	2	21/16
INI	ED A2		x	?	?	?	1	2	16
IND	ED AA		x	?	?	?	1	2	16
OUT (port),A	D3							2	11
OUT (C),reg	ED 01rrr001							2	12
OTIR	ED B3		1	?	?	?	1	2	21/16
OTDR	ED BB		1	?	?	?	1	2	21/16
OUTI	ED A3		x	?	?	?	1	2	16
OUTD	ED AB		x	?	?	?	1	2	16

SUBROUTINE CALLS, INTERRUPTS AND FLAGS

Source code	Op-code	C	Z	S	P/O	Ac	N	Bt	Ck
CALL label	CD							3	17
CALL cnd,lab	11ccc100							3	10/17
RET	C9							1	10
RET cnd	11ccc000							1	5/11
DI	F3							1	4
EI	FB							1	4
RST n	11sss111							1	11
RETI	ED 4D							2	14
RETN	ED 45							2	14
IM 0	ED 46							2	8
IM 1	ED 56							2	8
IM 2	ED 5E							2	8
SCF	37	1				0	0	1	4
CCF	3F	x				?	0	1	4
NOP	00							1	4
HALT	76							1	4

APPENDIX 2: Z80 Instruction mnemonics

Note: many of the following mnemonics can be used with a variety of addressing modes. Where these are not shown, consult Appendix 1.

Commonly used instructions

Source code	Meaning
ADC A	Add with carry to accumulator
ADD	Add
AND	Logical AND
CALL addr	Call subroutine at address
CP	Compare
DEC	Decrement
DJNZ	Decrement and jump if non zero
EX DE,HL	Exchange DE with HL
INC	Increment
JR	Jump relative unconditional
JR cond,addr	Jump relative on condition
LD reg,data	Load register immediate
LD reg,(HL)	Load register by data pointer
LD A,(addr)	Load Accumulator direct
LD HL,data	Load HL immediate
LD (HL),reg	Store register, using HL as data pointer
LD (addr),A	Store Accumulator direct
LD dst, src	Move source register to destination register
POP	Pop pr from stack
PUSH	Push pr on stack
RET	Return from subroutine
RLA	Rotate Accumulator left through carry
RRA	Rotate Accumulator right through carry
SBC	Subtract with carry (borrow)
SLA	Shift left arithmetic
SRL	Shift right logical
SUB	Subtract

Less often used instructions

Source code	Meaning
BIT b,(HL)	Compare bit b in memory
CPD	Compare Accumulator with memory. If Accumulator matches, set Z flag and decrement HL and BC. HL is address pointer and BC is byte count.

CPDR	As CPD but repeats until a match is found or byte count = zero
CPI	Compare Accumulator with memory. If Accumulator matches, set Z flag and increment HL and decrement BC. HL is address pointer and BC is byte count.
CPIR	As CPI but repeats until a match is found or byte count = zero
CPL	Complete Accumulator
DAA	Convert Accumulator contents to BCD form
DI	Disable interrupts
EI	Enable interrupts
HALT	Halt
IN	Input
IND	Input from I/O port to memory location (addressed by register C). Decrement registers B and HL. B is byte count and HL is address pointer
INDR	As IND but repeats until match is found
INI	Input from I/O port to memory location (addressed by register C). Decrement registers B and increment HL. B is byte count and HL is address pointer
INIR	As INI but repeats until a match is found or byte count = zero
JP addr	Jump unconditional
JP cond,addr	Jump on condition
LD A,(BC(	Load Accumulator using BC as address pointer
LD A,(DE)	Load Accumulator using DE as address pointer
LD HL,(addr)	Load HL direct
LD reg,(xy+d)	Load reg, indexed
LD rp,(addr)	Load register pair direct
LD rp,data	Load register pair immediate
LD xy,(addr)	Load index register IX or IY direct
LD (BC) or (DE),A	Store Accumulator secondary
LD (addr),HL	Store HL direct
LD (xy+d),reg	Store register, indexed
LD (addr),rp	Store register pair direct
LD (addr),xy	Store index register IX or IY direct
LD (HL),data	Store immediate to memory
LD (xy+d),data	Store immediate to memory, indexed
LDD	Load memory addressed by HL to that addressed by DE, decrement BC,DE and HL
LDDR	Repeat, as above until BC contains zero

LDI	Transfer byte from memory to memory. Increment HL and DE, decrement BC. HL is address pointer, DE contains destination address, BC is byte counter
LDIR	Same as LDI but repeats until BC = zero
NEG	Negate (two's complement) Accumulator
NOP	No operation
OR	Logical OR
OUT	Output
OUTD	Output from memory to I/O port addressed by register C. Decrement HL and B. HL is address pointer, B is byte counter.
OTDR	Same as OUTD but repeats until B = zer0
OUTI	Output from memory to I/O port addressed by register C. Increment HL and decrement B. HL is address pointer, B is byte counter.
OTIR	As OUTI but repeats until B = zero
RES	Reset bit
RETI	Return from interrupt
RL	Rotate left through carry
RLC	Rotate left circular
RLCA	Rotate Accumulator left circular
RR	Rotate right through carry
RRC	Rotate right circular
RRCA	Rotate Accumulator right circular
SET	Set bit
SRA	Shift right arithmetic
XOR	Logical exclusive or

Seldom used instructions

Source code	Meaning
ADC HL,rp	Add register pair with carry to HL
CALL cond,addr	Call subroutine conditional
CCF	Complement carry flag
EXX	Exchange register pairs with alternative register pairs
IM 0	Set interrupt Mode 0 (interrupt device places instruction on data bus)
IM 1	Set interrupt Mode 1 (interrupt device causes restart to address #0038)
IM 2	interrupt device performs an indirect call to any address. (Interrupt Vector 1 register supplies high byte address and interrupt device supplies low byte)

RET cond	Return from interrupt conditional
RETN	Return from non-maskable interrupt
RLD	Rotate Accumulator and memory left (BCD format)
RRD	Rotate Accumulator and memory right decimal
RST	Restart
SCF	Set carry flag
LD A,I	Load Accumulator from interrupt vector register
LD A,R	Load Accumulator from refresh register
LD I,A	Store Accumulator to interrupt vector register
LD R,A	Store Accumulator to Refresh register
LD SP,HL	Move HL to Stack pointer
LD SP,xy	Move index register X or Y to Stack Pointer.

Appendix 3: ASCII Character codes

Decimal	*Hex*	*Character*
32	20	Space
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	–
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N

79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	—
96	60	\
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	Delete

Appendix 4: Assembler directives

The following brief definitions are intended only for quick reference. For more extensive coverage, consult the Hisoft DEVPAC manual.

ORG expression
Meaning: Origin
Sets location counter to the value of 'expression'. (The address where the object code is to be assembled).

EQU expression
Meaning: Equate
Must be preceded by a label. Sets value of label to the value of 'expression'.

DEFB expression, expression, . . .
Meaning: Define Byte
Each expression must evaluate to 8 bits. Sets the value of each 'expression' in the list into the location defined by the current Location Counter.

DEFW expression, expression, . . .
Meaning: Define Word
Each expression can evaluate to 16 bits. Set the value of each two byte 'expression' in the list into the pair of locations (low byte first) defined by the current Location Counter.

DEFS expression
Meaning: Define Size
Reserves a block of memory, equal to the value of 'expression', by adding it to the current value of the Location Counter.

DEFM 'string'
Meaning: Define Memory
Places the ASCII codes of characters, enveloped within the quotes, directly into memory at the current value of the Location Counter.

ENT expression
Meaning: Sets the execute address of the object code to the value of 'expression'. Allows the code to be executed from within the assembler by the editor command 'R'.

Index

Accumulator 54
Action games 213
Adding BCD numbers 34
Adding HEX numbers 33
Addition and subtraction (8 bit) 92
Addition and subtraction (16 bit) 95
Addition and subtraction (32 bit) 98
Address bus 11, 12, 15
Addresses 13
Addressing limits 13
Addressing modes 64
Algol 2, 3
AMSOFT 6
Analogue computers 21
AND 35, 36
Arithmetic instructions 70
Assembler (use of) 41
Assembler directives 43
Assembler loading 40
Assembler options 45
Assembler output (making sense of) 43, 45
Assembler output columns 45
Assembler passes 42
Assembly code 5, 7
Assembly listing example 44

Bank switching 16
BASIC loader program (use of) 47
BASIC loader program 38, 39
BASIC ROM 16
BASIC/machine code hybrid programs 8
BCD 33, 35
Binary addition 24
Binary 6, 11, 22
Bit significance 24
Bit test instructions 78
Bit testing 116
Bits (ways of arranging) 23
Branch if equal 104
Branch if greater than or equal to 105
Branch if greater than 106
Branch if less than or equal to 106, 107
Branch if less than 105, 106
Branch if non zero 103
Branch if not equal 104
Branch if zero 103
Branch on sign 115, 116
Branch on zero (8 bit) 101
Branch on zero (16 bit) 102
Bytes 23, 66

CALL 48, 51, 77
Cathode ray tube controller 17
Clock cycles 66
Clock 15
Co-ordinate blocks 227
Code conversions 150, 154
Comparison instructions 78
Compiler 3, 4, 5
Conditional jump instructions 46, 76
Control bus 11, 13
CPC 464 block schematic 14
CRTC 17, 18

Data pointer (HL) 57
Data 13
Database 8
Debugging 4
Decimal to binary conversion 30
Decimal 7
Decrement instructions 75
Digital computers 21
Direct addressing 64
Disc operating system 19
Division (signed 16 bit) 133
Division (unsigned 16 bit) 132
Division (unsigned 8 bit) 129
Division 128
DOS 19
Dynamic allocation of memory 185

ENT 43, 50
EQU 44, 45, 46

Execution from basic 163, 175

Firmware routines (list of) 87
Firmware routines 213
Firmware 16, 86
Flag bits 54
Flag register 60, 67
Fortran 3
Frame flyback 219

GOTO 2

Hardware 10
Hex numbering limits 33
Hex to decimal conversion 32
Hexadecimal 6, 7, 13, 31
High and low state values 22
High byte 25
High level languages 3
HIMEM 19
HISOFT DEVPAC 6, 19, 39

IEE 74 standard 35
Immediate addressing 64
Implied addressing 58, 64
Increment instructions 74
Index registers 58
Indexed addressing 65
Input (decimal) 147
Input (hexadecimal) 151
Input (string) 139
Instruction set (presentation of) 66
Instruction set 12
Instructions (commonly used) 63, 68
Integrated circuit 10
Interpreter 3, 4, 5
Interrupt vector 62
Interrupts 17

Jump blocks 87
Jumps using direct addressing 84

Labels 39, 45
Line numbers 41
Load instructions 68
Logic gates 11
Logical instructions 72
Logical operations 35
Loop (double byte downcounting) 111, 114
Loop (double byte upcounting) 109, 112
Loop (single byte downcounting) 110, 112
Loop (single byte upcounting) 109, 111
Loop structures 107
Loop until key pressed 108
Low byte 25

Machine addresses 31
Machine code loader 6
Machine code 5
Memory map 19
Microprocessor 8, 11
Microprocessors (16 bit) 13
Microprocessors (32 bit) 13
Mnemonic code 5, 39, 45, 63, 65
Mode change 220
Mode zero 215
Multiplication (signed 16 bit) 123
Multiplication (unsigned 8 bit) 119
Multiplication (unsigned 16 bit) 121
Multiplication 118

Numbering system base 31
Numbers (double byte) 25
Numbers (signed) 26
Numbers (single byte) 24
Nybbles 24, 35

Object code (binary file) 174
Object code (execution from basic) 51
Object code (execution of) 50
Object code (general) 2, 3, 4, 45, 46
Object code (location independent) 186
Object code (saving) 49
Object code (self relocation) 185
One's complement 27
Operating system ROM 16
Operating system 16
Operation code (details of) 81
Operation codes 65
OR 35, 36
ORG 43
Output (decimal) 155
Output (hexadecimal) 157
Output (string) 143
Output (text) 144
Overflow 30

Pages 31
Palette latch 17
Parameter passing 161
Pascal 2, 3
PEEK 39
Pixels (encoding of) 216
POKE 39
Poking code into memory 5
Poking machine code into memory 38
POP 59
Program counter 54, 59
Program relative addressing 64
PUSH 59

Quartz crystal oscillators 11

Quicksort (RSX. rect. string array) 200
Quicksort (string array) 172
Quicksort algorithm 164
Quicksort as an RSX (string) 188

RAM based data files 200
RAM 2, 7, 15, 16, 17
Range limitations 92
Reading from memory 13
Rectangular arrays (efficient use) 200
Rectangular arrays (sorting of) 200
Rectangular arrays (storage details) 201
Refresh register 62
Register indirect addressing 65
Register pairs 57
Register source and destination rules 53
Registers 12, 53, 56
Relative jump bytes (counting of) 83
Relative jump instructions 102
Relocation (converting existing code) 194
Relocation problems 186
Relocation 175
Resident system extensions (RSX) 180
RGB 17
ROM (general) 15, 16, 17
ROM overlay 16
Rotate instructions 79
RSX (converting a subroutine to) 187
RSX (loading and testing) 208, 198
RSX (logging on) 180
RSX (producing a binary file) 199, 210
RSX (Quicksort rectangular array) 201

Screen memory (addressing of) 219
Screen memory 214
Shape (movement of) 220
Shape positioning 217
Shape tables 216, 229
Shift instructions 79
Sign bit 27
Sorting basic string arrays 163
Sound generator 19
Source code (deletion of) 49
Source code (hard copy) 49
Source code (listing) 42
Source code (loading) 48
Source code (renumbering of) 49
Source code (saving) 48
Source code 2, 3, 4, 45
Spaghetti programming 2
Stack (use of) 59, 114, 157, 160
Stack instructions 75
Stack pointer 59
Stack 19
String descriptors 172
Structure 2
Subroutine calls 77
Switching between basic and assembler 51

Translation software 3
Tristate logic 17
Two state logic 11, 17
Two's complement (circle) 28
Two's complement (double byte) 29
Two's complement (negative limit) 29
Two's complement (positive limit) 29
Two's complement (rule) 27
Two's complement 26, 33

ULA 15, 16, 17, 18
Unconditional jump instructions 77
Unsigned binary 30

Work space 2
Writing to memory 13

XOR 35, 36

ASSEMBLY LANGUAGE PROGRAMMING FOR THE AMSTRAD CPC 464, 664 & 6128

Although the BASIC language provided on the Amstrad machines is of an exceptionally high standard, there are obvious advantages, both in speed and memory economy, to be gained by the addition of machine code subroutines.

This book is intended to provide a grounding in Amstrad machine code programming. Although particular emphasis is given to the more commonly used Z80 instructions, the full instruction set is presented in an appendix, specially designed for quick reference.

The text is supplemented by numerous practical examples including several full length listings. These are laid out in standard assembly format, but a BASIC program is provided for direct loading of machine code bytes to help those readers who have not yet obtained an assembler.

ISBN 0-85242-861-8

ASSEMBLY LANGUAGE PROGRAMMING FOR THE AMSTRAD CPC 464,664 & 6128

ARGUS BOOKS

Document numérisé avec amour par

AMSTRAD

MÉMOIRE ÉCRITE

https://acpc.me/