BEST OF PCW

TEACH YOURSELF ASSEMBLER Z80

Paul Overaa

TEACH YOURSELF ASSEMBLER

Z-80

Paul Andreas Overaa

Century Communications London

ABOUT THIS BOOK

Teach Yourself Assembler provides a clear approach to structured programming through the use of Warnier diagrams. The use of the Z-80 processor and its instruction set are discussed in sufficient detail to allow the novice to start programming. All the main structures of a program are illustrated extensively with Z-80 and BASIC listings. General approaches to problem solving and to the real-life application of programs are considered at the end of the book.

A detailed tabulation of all the major Z-80 op-codes and their functions completes the book.

ABOUT THE AUTHOR

Paul Andreas Overaa works for a company of consultant analytical chemists in London, England. He initially specialized in gas liquid chromatography, a branch of physical chemistry, and it was through this that he became involved with data reduction techniques and microprocessor programming. Much of his published work centres around the use of language independent design techniques.

He is a Fellow of the Institute of Analysts and Programmers and a Licentiate of the Institute of Data Processing Management. His spare time interests are varied and include pyschology, mathematics, music and Yoga.

ALSO AVAILABLE from **Century Communications**
in association with *Personal Computer World*
Three new books about assembly language programming:

Teach Yourself Assembler: 6502 by Paul Overaa, a companion to this book based on this other very popular processor.

Assembler routines for the Z-80 and
Assembler routines for the 6502 by David Barrow, based on the very best of the long running ***PCW SUBSET*** series.

Copyright © Paul Andreas Overaa 1984

Based on material published in *Personal Computer World* magazine

All rights reserved

First published in Great Britain in 1984 by

Century Communications Limited
12-13 Greek Street, London, W1V 5LE

ISBN 0 7126 0549 5

Edited and produced working directly from the author's word-processor by NWL Editorial Services, Somerset, TA10 9DG

CONTENTS

Chapter 1	**INTRODUCTION** An introduction to the book and to the approach adopted throughout it	1
Chapter 2	**WARNIER DIAGRAMS** An explanation of Warnier diagrams in relation to program design	7
Chapter 3	**ASSEMBLY LANGUAGE** A discussion of assembly language and of the facilities offered by modern assemblers	19
Chapter 4	**THE Z80 PROCESSOR** An examination of the classes of instructions available for this processor	27
Chapter 5	**SEQUENCE AND REPETITION** Two of the basic building blocks of structured programming	37
Chapter 6	**ALTERNATION** The third basic building block of programming is examined	47
Chapter 7	**ADDRESSING** An explanation of addressing techniques	59
Chapter 8	**REPRESENTING NUMBERS** A description of the different techniques used for different types of numbers	79
Chapter 9	**DATA STRUCTURES** An introduction to commonly used data structures	93
Chapter 10	**SORTING AND SEARCHING** Some simple search techniques are examined before looking at tree structures and sorts	113
Chapter 11	**SOLVING YOUR PROBLEMS** A general approach to problem solving and program design	139
Chapter 12	**LINKING INTO BASIC** Two simple projects are described	147
Chapter 13	**WHERE TO GO FROM HERE** Where indeed?	157
Appendix A	**The Z-80 Instruction set listing**	167
Appendix B	**Assembler conventions**	221
Appendix C	**The ASCII Character set**	223
Appendix D	**The CP/M Operating system**	227
Appendix E	**Glossary**	231
Index		236

Understand the stillness that lies within you
and be content to know that
you are not alone

---

dedicated to

BOO

for very special help

1 SOLVING PROBLEMS

There are so many methodologies scattered around under the heading '*structured approaches to problem solving*', '*structured programming techniques*', and so on, that we are apt to look on new ideas and thoughts about how we should program with a certain contempt. Frequently such contempt is justified because writers often re-hash the work of others – using little more than a different terminology to hide the fact. This book collects together some important ideas that have appeared over the last few years and attempts to illustrate how these ideas can give you a foothold into the world of assembly language programming. Emphasis is placed on how these newer techniques are evolving to form a self-consistent framework of ideas applicable to both high and low level programming.

Let us start by examining some very general points, closely related to the field of computer programming – they are concerned with how we think and how we solve problems.

STRUCTURE, LOGIC AND CODE

It is generally accepted that the easiest way to learn about any new subject is to break it up into small manageable pieces. Each piece is then far less formidable and consequently far easier to get to grips with. Inherent in this idea is the implication that an ordered or 'structured' approach exists which enables our understanding of the lesser problems to be integrated into our understanding of the original more complex ideas and problems.

Once you have a computer (or access to one), instruction manuals and books explaining how to program your computer, and a certain amount of 'hands-on' experience, you will begin to feel ready to tackle larger problems – and this is when the difficulties arise. As often as not you find yourself searching through large amounts of coding in an attempt to locate a 'bug' which is preventing your program from working. If you are examining a program that you wrote some time ago the problem may be even more exasperating (especially if you did not document it properly). In even worse cases you find yourself trying to understand programs written by other people. You may well have come to the conclusion that either every programmer is a latent masochist – or that there must be a better approach to use.

The emphasis towards 'structured programming' is an important step in the right direction but it is not in itself a complete solution. This is because a serious fundamental error is continually made by both professional and amateur programmers alike. The difficulty in programming a computer to solve a particular problem consists of two very distinctly different parts:

- The inherent logical basis of the problem

- Writing the code in the language of your choice

The confusion between these two problems is one of the reasons why so many people run into problems as they start to tackle larger projects. It is also one of the factors that can make the difference between being able to learn assembly language and giving up; this next paragraph offers the key to overcoming 99 percent of your programming difficulties:

- Any envisaged use of computers to solve a problem requires that you find a logically correct solution *before* you make any attempt to to actually code your computer solution – *you should not try to solve the quite separate problems at the same time.*

The problems associated with tackling each part in turn have, time and time again, been found far easier to deal with than the difficulties incurred by approaches not making this distinction. The isolation of a logical program design produces a logical solution that is portable i.e. is independent from the computer hardware and software on which it will be implemented.

From a practical viewpoint it is obviously advantageous to find ways of solving problems and designing logical solutions that can produce good, efficient, well structured programs in any language you care to name.

We are now going to examine three ideas related to how we solve problems, and to how we react to difficulties, and we shall see how the concept and use of the ***Warnier diagram*** can reduce many problems to the status of an open book. These ideas are unusual material for an assembly language book, but their relevance should not be underestimated: learning to program in assembly language is more difficult than high level language programming and it is useful to know a little about the way in which we think and how we react when things go less smoothly than we might wish.

ENACTIVE, SYMBOLIC AND ICONIC MODELS

The way in which we approach a problem plays an important role in determining how successful or not our solution will be. In the last

twenty years much work has been done by psychologists to try to discover the basic mechanisms we use when we solve a problem, i.e. how we learn, how we conceptualize and abstract and in general what mechanisms we use to come to terms with our intellectual and physical environment.

Jerome Bruner has attempted to describe and characterize the ways in which young children react when confronted with a problem. He was able to identify three broad stages in the problem solving experience. The words used by Bruner – ***Enactive***, ***Iconic*** and ***Symbolic*** – can be thought of as keywords for a basic problem solving framework. This framework is applicable to adult as well as children's patterns of thought.

- The *Enactive* stage relates to the use of physical models and the ability and confidence to manipulate them. One of the characteristics of this enactive level is the inability to describe the situation, i.e. the inability to communicate effectively without resorting to actual demonstration.

- The second, *Iconic*, stage describes the use of diagrams or pictures to represent the 'enactive elements' of the problem. This has been called the 'iconic' stage and is sometimes seen as the initial stage of abstraction, i.e. of separating the physical or real problem into a 'modelling situation'. Such a model will hopefully embody all the enactive elements of the problem in a form that is easier to translate into totally abstract form.

- The third, *Symbolic*, stage describes the use of signs and symbols, previously defined, to produce an abstract version of the problem. This characterizes the 'symbolic' level of confidence in problem solving.

Mathematics is typical of symbolic abstraction and it is commonly recognised that difficulties associated with learning and understanding mathematics frequently stems from a lack of confidence in symbol manipulation.

In children these stages can be identified by the way that simple problems are tackled. Of equal importance is how the approach changes as a child gets older: given a dozen bricks a very young child, if asked how many he would get if he had to share his bricks with two other children, will resort to physically (*enactively*) sharing the bricks. At a later stage he might solve such a problem by drawing three boxes and placing dots in them to represent bricks. He will be able to deduce from his *iconic* model that each child will receive four bricks. Later still in his development his confidence at the *symbolic* level will enable him to write '12 ÷ 3 = 4' without hesitation – and to know that the answer produced *symbolically* is as valid as both the enactive and iconic results.

In many situations these three levels of confidence occur simultaneously; they should not therefore be thought of as being physically distinct. The distinction to make is that the stages are conceptually different. We will often be able to look at particular lines of reasoning and identify problem areas as being 'symbolic' as opposed to 'iconic' (or whatever).

This framework is equally recognizable in adult thinking and the various levels of confidence can often be identified. A point that is important is that when we have difficulties in tackling a problem we frequently fall back to a lower level of problem representation in an attempt to achieve a better understanding.

As an example consider how many times you have been presented with a mathematical problem to solve in which you plunged straight in with some symbolic argument only to find you got 'stuck' and rapidly resorted to a graph or diagram – an 'iconic model' – in order to get a better understanding of the problem itself.

These ideas produce some interesting generalities which have implications of particular benefit to us in our quest for better methods of designing and writing computer programs.

Firstly, when you solve programming problems you are frequently solving other people's problems. You may very often need to explain your solutions and your lines of reasoning to others and there is a need to ensure proper communication of your ideas (often to non-technical people).

Secondly the problems you examine will often be ill defined or imprecisely defined. Frequently restrictions will be added to the problem whilst you are in the middle of finding a solution and the problem will change.

During all your programming you will regularly and inevitably encounter a number of quite severe difficulties. If you are working at a purely symbolic level you may conclude that some particular difficulty is insurmountable. Providing you have an 'iconic model' to fall back on you are more likely to come to terms with the new restraints.

The ways in which we describe a problem are important to us as we attempt to conceptualize and solve problems. The description is also important because of the ease or difficulty with which we can convey our ideas and thoughts to others.

WARNIER DIAGRAMS

One of the techniques which capitalizes on the above ideas is the Warnier diagram. The power of using such diagrams to design programs is due in part to the separation of the logical from the

practical difficulties. In addition to this the diagrams provide an iconic level with which to examine a problem.

The next chapter shows you what a Warnier diagram is and introduces you to the way it can be used to describe a computer program. We use these diagrams no matter what language we program in because they help to define and clarify the fundamental logical problems of the program design – regardless of the code which will be written to execute it. We hope that by the time you have finished this book you also will appreciate their usefulness, not only in relation to learning Z-80 assembly language but to programming and system design problems in general.

The ideas that have been discussed have implications not only in the field of programming but in thinking itself. In the Warnier diagram we have a technique which enables us to draw a picture of the logical structure of a problem. We can successively redefine our thoughts, change the problem, add or remove restrictions, all while maintaining a diagrammatic version of the current solution ready to be translated into any language we choose to use. Throughout this book you will see Warnier representations of various problems and we hope you will, as we have done, realize that the work of Warnier goes far beyond the realms of programming and system design, he has in fact given us the techniques for analysing our thoughts and organizing the contents of our minds in a way that enables us to document those thoughts whilst we examine problems. We can illustrate our progress diagrammatically and use the diagrams produced to successively refine our ideas etc. The techniques actually help us think about the problems we examine.

If you are a newcomer to micro-computing then take heart. Although some of the ideas may take a certain amount of time to digest they are basically simple. Be patient and think about the concepts. Apply them to problems of your own choosing and you will achieve a real and useful understanding.

The Z-80 PROCESSOR

We know from first hand experience that too much dependence on any particular processor often tends to result in the teaching of selected tricks based on the particular facilities that the processor provides. Whilst obviously it is important to use such techniques to utilize the available facilities of a particular processor, we feel that teaching underlying general principles is far more important.

We considered that the concept of working with a 'hypothetical processor' also leaves much to be desired. You only learn about assembly language programming by doing it and you can't easily run programs based around a hypothetical processor. We will use many of the concepts that you know about already from languages like

BASIC, and re-apply them to assembly language – we will also try, as far as possible, to avoid involvement with the I/O problems of specific machines.

The 'chip' we have chosen to work with is the *Zilog* Z-80 microprocessor. This is used in the *Sinclair* ZX-81 computer, the *Osborne*-01 and many other popular computers. So if you want to step into the world of assembly language programming but have been worried about the difficulties then please join us now as we take our first steps.

2 WARNIER DIAGRAMS

A program design approach must satisfy several criteria; it must be able to produce results of consistently high quality, it must be rapid, it must allow for easy 'program maintainence', and it must be simple and straightforward to use. You will be pleased to hear that a technique already exists which – to a large extent – satisfies all of the above objectives.

The pioneering work of Jean Dominique Warnier (*References 1 and 2*) in France represents a major step forward in the design of logically structured programs. The use of the 'Warnier diagram' has been recently publicized in some of the works of Kenneth T. Orr (*References 3 and 4*) and others in the United States.

SEPARATING OUT THE PROBLEM

Before looking in detail at the ideas involved it is important to emphasise that we are aiming to obtain solutions to problems that are completely independent of the computer or languages you use. Such factors will affect how you 'code' your solution, but they should not usually influence your logical solution to a design problem. The same techniques are applicable whether programs end up being coded in high level languages such as BASIC, PASCAL or COBOL, or coded in low level languages such as Z-80 assembly language. When you program in a high level language it is frequently possible to write short programs without explicitly designing your program. When programming with assembly language, the design stage becomes not just more important – it becomes vital. It provides a means of separating the *logical problems of design* from the *practical problems of coding* and by doing so enables you to tackle your programming in coherent stages.

Essentially a Warnier diagram is a set of 'curly brackets', that defines particular groups of operations and the order in which they should be performed. The easiest way to show you about these diagrams is to take some examples.

EXAMPLE A

Imagine we wish to produce a report, consisting of details held on a computer file on disc or tape. The Warnier diagram of the basic problem is shown in figure 2.1.

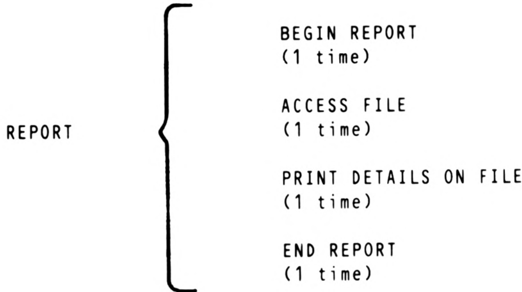


Figure 2.1: **Warnier diagram of the essential characteristics of a simple report generator**

The bracket is read from top to bottom and describes a procedure or group of operations that we have arbitrarily called REPORT. Underneath each item we have identified how many times the item is to be performed and, with this, our first diagram illustrates the essential features known about our problem.

Do we know anything more about our problem? Can we think of any information that could be relevant? Well, we know that:

- Computer files need to be *opened* before reading and *closed* once the read operation is complete.

These details could therefore be added to the diagram. To enable us to explain some further conventions used with Warnier diagrams let us first add a minor complication to the problem – let us suppose:

- The user wishes to access a file of his (or her) own choosing and to obtain a printed report of the details on the file.

- The specified file may not exist, and, if this is the case the user should be informed.

These changed or altered requirements can be represented by a more detailed Warnier diagram as we shall now show.

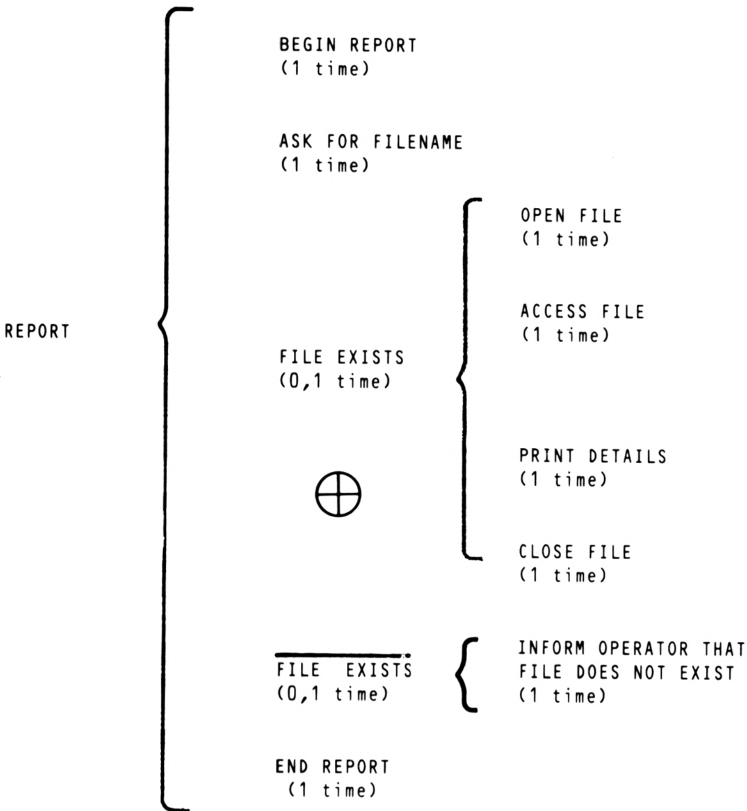


Figure 2.2: **Some new restrictions added to Figure 2.1**

Figure 2.2 shows, in Warnier diagram form, the requirements of our problem as it is at the moment. We are using the convention that the logical opposite of a statement is written by placing a bar over it:

$\overline{\text{FILE EXISTS}}$ means FILE DOES NOT EXIST

We are also using a sign ('⊕') to separate mutually exclusive operations (sets of operations which will not occur together). In our example the file will either exist or it will not exist – so only one of these two operations would be performed at any one time. In some cases (where one of two or more alternatives exist) there will be sets of operations which may not be performed or which may be performed once, or many times. This is shown on the diagrams as (0,1 time), (0,N times), etc.

The conventions we have used so far are in fact the only ones you will need for the majority of problems that you will encounter. Let's collect them together for convenience:

- Brackets are used to define sets of operations.

- Brackets are read, and performed, downwards within any one 'level'. The item at the top of the bracket is performed first, the item at the bottom performed last.

- The logical opposite of a statement can be written as the original statement with a bar drawn over it.

- Brackets written to the right of a statement indicate the operations to be performed if *that statement is performed.*

- *Underneath each item or statement we indicate the number of times the operations should be performed.*

Using these conventions we can express in English exactly what figure 2.2 tells us; we are dealing with a certain procedure, called REPORT that starts by asking for the name of a file. If the file exists then it is opened, accessed, the details printed, and then the file is closed. If it does not exist then the operator is informed of the fact. Remember that if the file does exist then it is the group of actions (subset) shown to the right of the label FILE EXISTS that are performed.

To appreciate the elegance and speed with which these diagrams can accommodate changing requirements let us place some further restrictions on our problem: within our 'hypothetical system' are files containing sensitive data (personnel data, wages, medical records, etc.). Such data must be protected from unauthorised access and users are therefore issued with access code numbers, so that examination of sensitive files is restricted to those users with the proper authority. If unauthorised attempts to access this data are made the computer should inform those in charge of system security.

Let us first consider the new constraints in isolation. We need to check whether the file specified by the user is a restricted file, if it is we must ask for the user's code number. If the code is correct then we allow access, if not we inform the system security of an attempted illegal access.

```
        ┌ BEGIN REPORT
        │ (1 time)
        │
        │ ASK FOR FILENAME
        │ (1 time)       ┌ BEGIN FILE EXISTS
        │                │ (1 time)          ┌ ASK FOR CODE
        │                │                   │ (1 time)
        │                │                   │
        │                │ FILE RESTRICTED   { CODE CORRECT   ACCESS FILE
        │                │ (0,1 time)        │ (0,1 time)     (1 time)
        │                │                   │
        │                │                   │                PRINT DETAILS
        │                │                   │      ⊕         (1 time)
        │ FILE EXISTS    {                   │
        │ (0,1 time)     │                   │               ┌ INFORM
        │                │       ⊕           │ ____________  │ SECURITY
        │                │                   │ CODE CORRECT  { ILLEGAL
        │                │                   │ (0,1 time)    │ ACCESS
REPORT  {                │                   └               │ (1 time)
        │                │                                   └
        │      ⊕         │                   ┌ ACCESS FILE
        │                │ ________________  │ (1 time)
        │                │ FILE RESRICTED    {
        │                │ (0,1 time)        │ PRINT DETAILS
        │                │                   └ (1 time)
        │                │
        │                │ END FILE EXISTS
        │                └ (1 time)
        │ ___________    ┌
        │ FILE EXISTS    │ INFORM USER FILE
        │ (0,1 time)     { DOES NOT EXIST
        │                │ (1 time)
        │                └
        │ END REPORT
        └ (1 time)
```

Figure 2.3: ***A hierarchy is forming within the revised problem***

The diagram in figure 2.3 shows the Warnier form of our new requirements. Notice that as we redefine the problem and add more detailed restrictions we do not have to rearrange the complete diagram, as one frequently needs to with flowcharts, etc. All we do is superimpose the new details and restrictions on to the existing diagram structure. The diagram is actually growing as we successively modify and redefine the known details of the problem. Additionally the diagram is documenting and expressing the logical requirements of the problem in a way that will make the transition to a computer language equivalent remarkably simple.

The ability of the Warnier diagram to display, help formulate, and to grow with the changing logical requirements of a problem, as that problem is examined, is of great importance. Once the quite simple conventions have been learnt these diagrams can be read just like the written English equivalent but, unlike the written English form, a Warnier diagram contains within its deceptively simple notation, the complete solution to coding of the problem.

We will see several examples of how this is achieved in assembly language in later chapters but for now figure 2.4 gives an example using a 'pseudo BASIC' type of code to show the general idea. The secret of converting a Warnier diagram into a finished program lies in regarding each bracket involving more than one operation as a subroutine. There are certain exceptions to this general statement but these will become apparent during the course of the book.

```
* ===================================================================
        P S E U D O - B A S I C - R E P O R T - M O D U L E
* -------------------------------------------------------------------
INPUT NAME OF FILE
IF FILE EXISTS THEN GOSUB 'FILE EXISTS'
                                ELSE PRINT 'FILE DOES NOT EXIST'
RETURN TO CALLING PROGRAM
* -------------------------------------------------------------------
REM SUBROUTINE..........FILE EXISTS
IF FILE IS RESTRICTED THE GOSUB 'RESTRICTED FILE'
                                ELSE GOSUB 'ACCESS'
RETURN
* -------------------------------------------------------------------
REM SUBROUTINE..........RESTRICTED FILE
INPUT SECURITY CODE
IF SECURITY CODE=CORRECT CODE THEN GOSUB 'ACCESS'
                                ELSE GOSUB 'ILLEGAL ACCESS'
RETURN
* -------------------------------------------------------------------
REM SUBROUTINE..........ILLEGAL ACCESS
WRITE TO I/A LOG FILE THE TIME OF ATTEMPT AND THE ACCESS CODE
PRINT 'THIS IS A RESTRICTED FILE - PLEASE MAKE NO FURTHER ATTEMPTS'
RETURN
* -------------------------------------------------------------------
REM SUBROUTINE..........ACCESS
THIS WOULD BE A ROUTINE TO ACCESS THE DATA IN THE FILE AND DISPLAY
ON TERMINAL OR PRINTER ETC.
RETURN
* ===================================================================
```

Figure 2.4: ***Pseudo-BASIC code for Example A***

EXAMPLE B

We wish to design the basic structure of a routine that collects characters from a keyboard device. If the character is a carriage return (i.e. ASCII 13) then we should leave the routine, if it is

another control character then an appropriate control character subroutine should be performed. If the character is not a control character then it should be passed to a printing routine to display it on a VDU or other output device.

Let us quantify what we know about the problem in terms of the sort of operations we may need to perform. We will have to input a character, possibly using an input routine available within the operating system. We must also make some type of check to see if it corresponds to a control character (note that for our purposes we shall regard a control character as one with an ASCII value of less than decimal 32). Additionally we will need some means of printing characters and again this may be a facility provided by the operating system. Let us draw a Warnier diagram to indicate the objectives we can recognize so far.

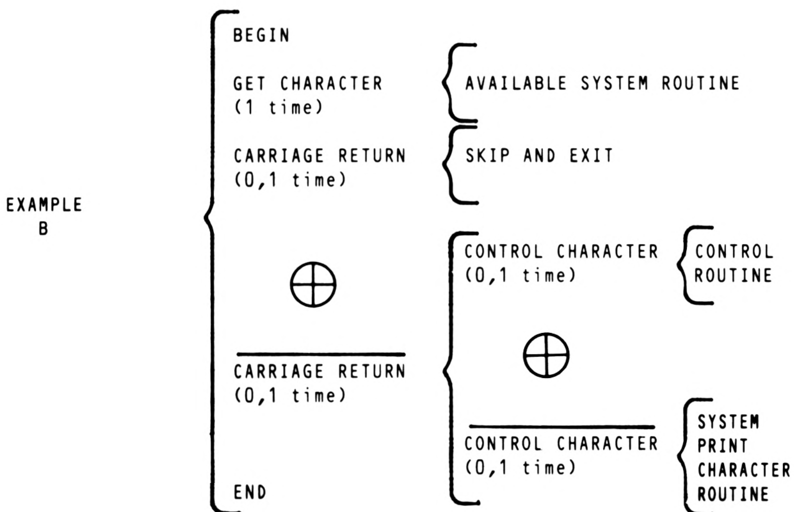


Figure 2.5: ***First Warnier diagram for Example B***

Figure 2.5 shows our first attempt at describing the problem. The diagram implies that we can perform a test that will indicate whether a given input character is a carriage return or not, additionally it implies that we can test to see if a character is a control character or not. We should be fairly happy with this initial diagram because we know that all computer languages provide the type of testing we

would need to use. In the BASIC language we are, for example, able to use statements such as:

```
IF ASC(X$)=13 THEN ...
```

and

```
IF ASC(X$)<32 ...
```

to perform the necessary tests.

At present the Warnier diagram does not indicate that we collect anything more than one character by performing the illustrated operations. It is necessary in practice to perform the operations in figure 2.5 any number of times from 1 to N times, depending on when the user supplies a carriage return character.

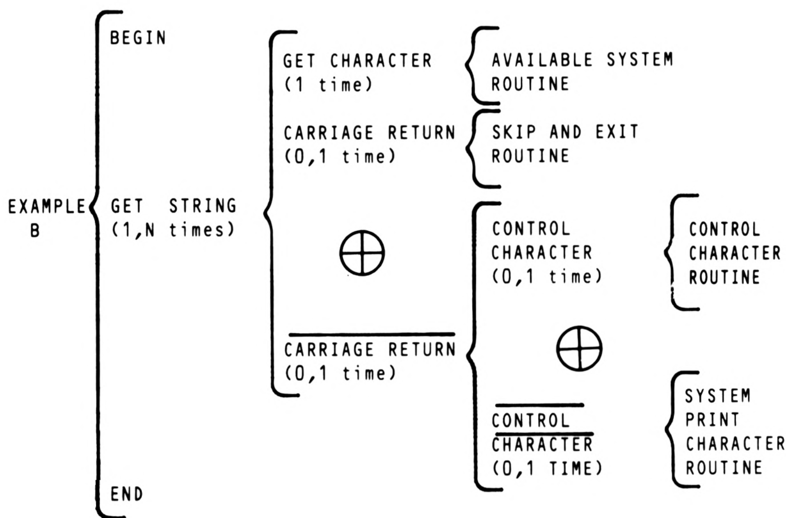


Figure2.6: ***Expanded Warnier diagram for Example B***

Figure 2.6 explicitly shows that we perform the operations indicated in figure 2.5 at least once, and up to a maximum of N times. The labels we use are, of course, arbitrary, but it is obviously advisable to use meaningful English expressions since this enables the diagrams to be easily understood.

We now have an accurate representation of the problem we are dealing with let's see how we can continue to redefine parts of the problem and expand the corresponding parts on the Warnier diagram. Let us suppose that the control characters detected are

going to be used to perform the operations shown in figure 2.7. We will, on the basis of the ASCII value of the control character, perform *one* of the routines listed.

ASCII code	Operation to be performed
8	Move cursor to *Left*
16	Move cursor to *Right*
10	Perform a *Line Feed*
9	Perform a *Tab*
11	Move cursor *Down*
12	Move cursor *Up*
OTHERS	Take *no action* (i.e. ignore them all)

Figure 2.7: **Actions associated with the control characters**

These operations are a more complex example of the mutually exclusive operation sets mentioned earlier. In such cases we cannot use the bar notation since many alternatives exist. Instead we show the options using their respective names and we use the ⊕ sign to indicate that each 'operation subset' is mutually exclusive. Figure 2.8 shows how we represent this in Warnier diagram form.

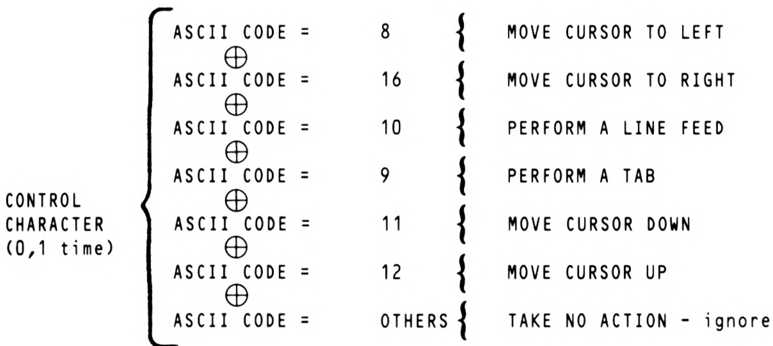


Figure 2.8: ***Warnier expansion of the CONTROL CHARACTER statement***

Let us now make an alteration to the control character's routine by creating some further assumptions. We suppose that if our hypothetical user presses a control key that serves no apparent purpose then either a simple error has been made (the user has pressed the wrong key) or the user is under the impression that the control key pressed serves some function which it does not, in fact,

perform. In either case we may, from a practical point of view, decide to provide some means of informing our user that a 'useless' or 'unsupported' key has been pressed.

We may assume that the VDU screen has one or two lines available for comments and for programs to use when collecting responses such as input from the user. We also must assume that the remainder of the screen contains information that must be preserved, so we cannot simply print a menu of control character options on to the screen. What criteria can we identify?

- We will need space on the screen to display a menu and will therefore need to save the existing contents of the VDU screen somewhere.
- We will need to ascertain whether the 'user' actually needs a menu or whether he or she will quickly realize that a wrong key has been pressed by mistake.

We first consider the new restrictions as a discrete subset of operations, i.e. we concentrate on these new requirements. Once we have created a suitably structured diagram concerning the new constraints we can then superimpose it on the original diagram in figure 2.8.

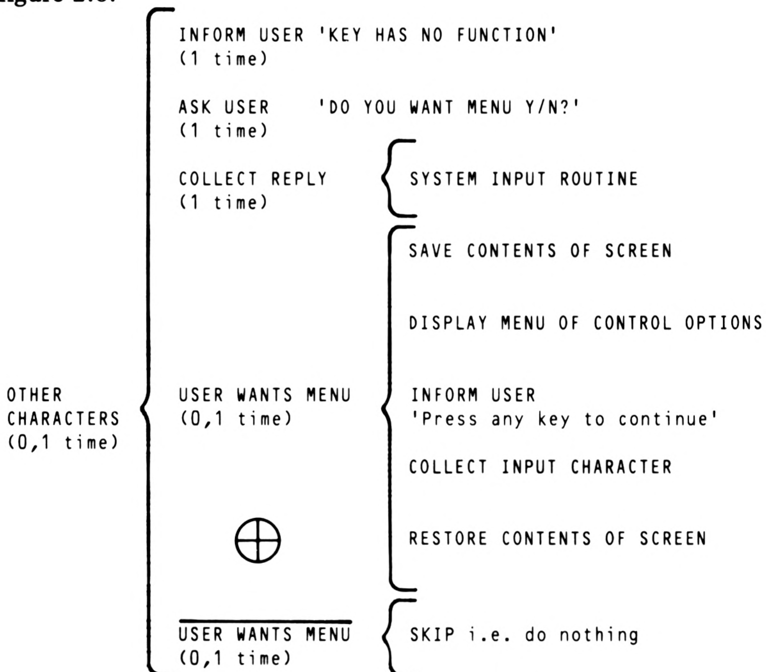


Figure 2.9: ***New restraints added to Figure 2.8***

The diagram in figure 2.9 shows our latest requirements in Warnier diagram form. Convince yourself that we have expressed the known additional details in a suitable manner, then look at figure 2.10 which shows the whole of the control character description including the current additions.

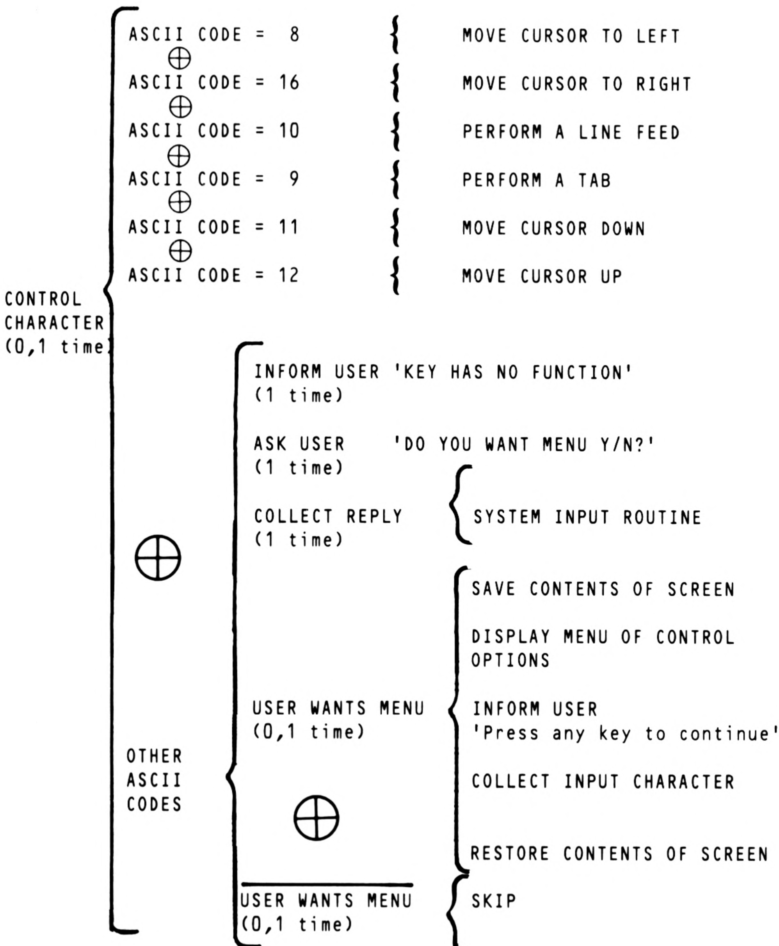


Figure 2.10: ***The final control character diagram***

We could continue to expand other statements to provide further detailed analysis of the problem. As we do so we reach a point where it is possible to say 'Yes, the operations we are describing in the lower levels of the diagrams (the right-most levels) are easily capable of being coded directly in the language I have chosen to use!' In

practice we reach this point sooner with high level languages than with assembly languages because more complex operations are supported. The relevant point to make is that the general principles are the same – the only difference is that when you analyse problems that will be coded in assembly language you will need to carry the analysis further.

FINAL WORD

The ideas discussed in this chapter have implications not only in the field of programming, but in thinking itself. We have a technique that enables us to draw a picture of the logical structure of a problem. We can successively redefine our thoughts, change the problem, and add or remove restrictions whilst maintaining a diagrammatic version of the current solution which can be translated into any language we choose to use. During the book you will be given the choice of examining the Warnier representations of various problems and we hope you will, as we have done, realize that the work of Warnier goes far beyond the realms of programming and system design, he has in fact given us the techniques for analysing our thoughts and organizing the contents of our minds in a way that enables us to document those thoughts whilst we examine problems. We can illustrate our progress diagrammatically and use the diagrams produced to successively refine our ideas. In short the techniques actually help us think about the problems we examine.

Reference 1:
Warnier, Jean Dominique – Logical Construction of Programs *(3rd Edition, translated by B.M. Flanagan)*
New York: Van Nostrand Reinhold Co. 1976

Reference 2:
Warnier, Jean Dominique – Logical Construction of Systems
New York: Van Nostrand Reinhold Co. 1981. ISBN 0–442–22556–3

Reference 3:
Orr, Kenneth. T. – *Structured Requirements Definition*
United States: Ken Orr and Associates, Inc. 1981.

ISBN 0–9605884–0–X

Reference 4:
Orr, Kenneth. T. – Structured Systems Development
United States: Yourdon Inc. 1977. ISBN 0–917072–06–5

3 ASSEMBLY LANGUAGE

For many computer hobbyists, being able to write programs in an 'assembler language' is an ultimate goal. The mystique of writing such programs is reinforced by the frequent publication of listings written in strange and wonderful symbols reminiscent of ancient Babylonian . . .

Many writers try to explain the concepts behind these languages in a manner that subsequently reinforces the idea that assembly language programming is different, complex, and difficult to understand. The result is of course a foregone conclusion – a lot of people rapidly decide that assembly language programming is going to be far too difficult for them to learn.

We want to show you that a knowledge of assembler can be a very real asset to your repertoire of computing skills and that it can be learnt without having to throw away all those principles of good programming that you developed while using higher level languages such as BASIC or PASCAL.

WHAT IS AN ASSEMBLY LANGUAGE?

Is an assembly language the same as 'machine code'? Is it the language that the microprocessor understands? Are all assembly languages similar to each other? Let us take things right from the start and discuss exactly what an assembly langauge is.

A *microprocessor* is a sophisticated logic device able to perform a substantial number of predefined functions. These functions are determined by the design of the 'chip' itself. On their own these individual functions are neither complicated, nor in fact particularly useful, but when combined in an appropriate order then the microprocessor becomes an exceedingly powerful and versatile tool.

There are many different processors available and the *architectures* of the various chips (their internal design) vary quite considerably. Certain generalizations can however be made to enable you to appreciate the essential similarities from an overall point of view.

As well as the set of operations that it can perform, a microprocessor also has available a set of *internal registers*, places where it may store data and other regularly needed items. It will have some registers

assigned for specific functions and others of a more general nature. Instruction sets are designed not only to utilize the internal facilities that the chip itself provides, but also to allow the use of additional memory chips. In this way the microprocessor is able to transmit and receive information – it has the ability to *communicate* with other devices.

Unfortunately the language a microprocessor understands – the *machine language* – is that of *binary numbers*, it identifies the various functions that it can perform in relation to predefined patterns of zeros and ones. The *Zilog* Z-80 processor will interpret the binary number 11000011 as an instruction to perform a jump from its current memory location to a different memory location, whose position or *address* would, in practice, be provided by the two binary numbers immediately following the instruction. It is possible to program a computer using nothing but such codes – and there are even certain occasions when it is necessary!

Such programming is, however, prone to many problems such as transposition errors and the like. To a certain extent the problem can be eased by using a *hexadecimal* or an *octal* numbering system. The above example for the jump instruction of a Z-80 processor changes from 11000011 in binary to a simpler C3 in hex. Not exactly a mind shattering improvement, but it's definitely going in the right direction.

A program written in a hexadecimal form becomes immediately unrecognisable to the processor, so it is always necessary to *convert* any such program into a binary form before it can be recognized as a set of instructions.

Early on in the development of the computer it became apparent that to write even moderate size programs in binary, hexadecimal or octal form was about the best working definition of masochism that you would be likely to find. The concept therefore developed of creating a language that used the same operations but with each of them being given a simple (and comprehensible) name. It was in this manner that *assembly language programming* was born.

The word used to describe the instructions is *mnemonic* – pronounced nem-mon-ik – literally a *memory jogger* or aid to memory (we have always felt that the individual who selected the word *mnemonic* was having a private joke against the rest of society). The mnemonic for the Z-80 jump instruction used earlier is JP, which you'll surely agree is a useful improvement.

By writing a computer program in such a language, we are attempting to make it more comprehensible to ourselves. We do not however make it any more comprehensible to the poor computer, and so it is necessary to translate our mnemonics into the binary form that the computer will be able to interpret. It can be done by

hand or a suitable computer program can do it for you. The name given to a computer program that performs this translation (or most of it) is an *Assembler*.

STANDARD MNEMONICS

The choice of mnemonics to be used with a particular type of processor is in fact an arbitrary one. They are selected and recommended by the manufacturer of the chips themselves as an aid.

In the same way that different software houses write their BASIC interpreters and compilers around various 'standard' facilities but still end up producing versions of BASIC slightly different from each other, so manufacturers of microprocessors also design according to their own ideas and idiosyncrasies. Thus each type of microprocessor will have its own characteristics and methods of implementing the functions it provides. It will also have its own set of mnemonics. Such mnemonics are peculiar to the processor or the series of processors that a manufacturer produces and thus the mnemonics that your computer will require will be dependent on the chip that your computer is designed around.

It would be perfectly possible, if you felt that the manufacturer had chosen their mnemonics unwisely, to devise your own set. You would however pay for such a privilege by having to either assemble your final programs by hand, or by having to write your own assembler. However, modern day assemblers also do far more for you than simply act as a mnemonic converter and we will look at some facilities provided later on.

PREPARING SOURCE CODE

In general the source code – the text version of the program that you write – would be prepared using some form of text editor. The source code may temporarily be stored on tape, on diskette or it may simply remain in memory. If it is stored at all it is usually as a simple ASCII text file.

Some assemblers have a 'resident' editor present as part of the assembler package itself. One such example of this arrangement is the *ZX.ASZMIC ROM* which actually replaces the *Sinclair* ZX-81 operating system to provide a quite useful Z-80 assembly language programming environment. You use an editor part to prepare your program in source code form, then assemble the program afterwards.

Other assemblers are completely separate. With these, usually on larger computer systems, you use a separate text editor or word processor to produce your source code. In these cases the assembler is used quite separately to operate on the ASCII files created by the text editor.

SYNTAX

Each assembler will have its own rules or *syntax* requirements, just as say BASIC and PASCAL have their own requirements. You need to be aware of what your assembler can cope with and to what extent, if any, you can deviate from those requirements.

Nowadays the average assembler will flag errors in much the same way as a BASIC interpreter does. Syntax and other common errors are checked for and identified. Most assemblers are, however, quite easily led astray. This being so, it is usually important to look at the first error that is shown very carefully since this one will often cause the assembler to mis-interpret succeeding statements, even though they are correct when considered in isolation.

Lines of an assembler listing

A line in an assembly language program can be divided into three regions or *fields*:

- A label field
- An instruction field
- A comments field

The first and last fields are optional. Your assembler will have fixed rules for identifying the individual fields and you will find these in the documentation provided with your assembler.

Here is an isolated line from an assembly language program:

```
CHECK$CHARACTER:   CP  CARRIAGE$RETURN   ;end of input?
↑                  ↑                     ↑
Label field        Instruction           Remark or comment field
(Optional)         Field                 (Optional)
```

Inclusion of comments within the program

It is possible to include comments within an assembly language program to make it more understandable. In BASIC you will be familar with the use of REM statements. With these you can add remarks that are effectively ignored by the BASIC interpreter. Assemblers vary in how they identify a 'Remark', the standard assembler provided by most CP/M systems will ignore any line that starts with an asterisk '*'. It will also ignore comments placed after a specified ';' delimiter character. Most BASICs have similar facilities for 'end of line' remarks. With a high level language you can often work out what a program does even if it has not been reasonably documented – *the same is not true of assembly language*.

One of the fundamental problems with writing assembly language is that it *is* difficult to analyse. It is difficult because any inherent structure present in the routines is often not obvious unless you wrote the code and know what to look for.

Because of this you may often have problems when you examine other people's programs. If you start right from the very beginning by making the maximum use of internal remarks you will at least avoid creating additional problems for yourself. You may think that you fully understand a program at the time you write it. You probably will at the time but will you next week, next month, next year? Make the most of a lesson that most of us have learnt the hard way and don't take chances – *document to the maximum extent that time (and your assembler) will permit.*

- The importance of placing understandable comments within an assembly language program cannot be over-emphasised.

Labels

Most assemblers allow you to use long meaningful names for specific locations in memory. This means that instead of having to remember that location $0A3F_{hex}$ is the start of some subroutine that checks characters collected from a keyboard, you can use a label `CHECK$CHARACTER` in the label field of the first instruction of that subroutine. The assembler will add the label to its internal *symbol table* and you will then be able to use the label to reference that particular routine. (Some assemblers require that you place a delimiter character, often a colon, immediately after a label definition so that the assembler can distinguish it from the instruction field, as in `CHECK:`).

The EQUate Directive

Another facility provided by modern day assemblers is that of the equate directive. This enables names to be assigned to numeric values. The use of such labels does not affect the code that the assembler will finally produce, and since it is a function of the assembler, *not* of the processor, the equate directive is known as a *pseudo operation*. It is especially useful for defining many of the common ASCII characters.

By placing the following statements at the start of your assembly language program you cause the assembler to include these 'definitions' to its internal symbol table.

```
CARRIAGE$RETURN         EQU     13
SPACE                   EQU     32
```

By using such definitions you will make your programs more readable to yourself and to others.

Other pseudo-ops

In addition to the above aids, the assembler will provide pseudo-operations that enable you to define specific areas of memory as reserved space, and to place certain constant values into the assembled program for you. You can, using a pseudo-op usually called an ORG directive (short for ORiGin), select whereabouts in memory your program is to start.

It is unwise to list and explain all the facilities available – for several reasons: firstly many functions will not make much sense until you have written some assembly language programs and, secondly, many of the functions are just icing on the cake and in the early stages their explanation simply causes undue confusion. We have given you the basic idea of what an assembly language is, of what an assembler actually *does* and of the type of help that modern assemblers can provide. We will add to this basic knowledge when it is appropriate.

OPERATING SYSTEMS

Before a microprocessor can perform any useful function, it needs a means of getting its data, it needs somewhere to send the output it produces, it needs additional memory, and it needs some means of co-ordinating all these various items, so that things happen at the right time and at the right place.

Unfortunately these very essential needs turn out to be the most complicated and time consuming to program. It is necessary to be familiar with the computer hardware used (and its idiosyncrasies), and with the technical details of the computer's design.

Routines which perform such functions are collectively known as the computer's *operating system*. It is neither necessary nor advisable for each manufacturer to design their own operating system and many micro manufacturers have realized the benefits of using widely available operating systems such as CP/M. These provide a quasi-standard environment within which to work. The object of such systems is to attempt to isolate you from the hardware-dependent nasties that present themselves when you start trying to get information to and from various devices. An operating system is to a large extent judged by its ability to isolate the problems of Input and Output (I/O) from a user.

There are unfortunately many different operating systems available. This is especially true of the smaller cassette-based home computers. With larger diskette based machines, CP/M provides a very welcome

anchor point which isolates you from the worst of its insides through a protocol-based function *interface*.

Smaller machines, in general, do not achieve this type of isolation. They nevertheless usually provide accessible routines which may be used by your programs to simplify many operations. It is not possible to be familiar with all of the different operating systems that are available. We will therefore try, as much as possible, to avoid reference to specific computers. This means that you will, when the time arrives, have to delve into your own computer's manuals for certain of the details you will need. By the time we get to this stage though we will hopefully have given you sufficient grounding in the general principles for you to know what you will be looking for.

NOTES

4

THE Z-80 PROCESSOR

The purpose of this chapter is to acquaint you with the various details about the Z-80 in order to help you build a mental picture of the processor. We can do this without having to delve into the world of electronics by using a simplified *logical model*. Many of the concepts we will discuss in this chapter are in fact applicable to many such models of other microprocessors.

We define a *register* as a place within a processor that can hold binary information. With 8-bit processors we talk of an 8-bit 'word length' and this is called a *byte*. The 8-bit registers in our processors can therefore hold one byte of information.

For our purposes a microprocessor can be regarded simply as a device that has a set of internal registers, some having specific uses, and a set of available instructions. The instructions enable data to be manipulated both within the processor and between it and the external memory.

Before a microprocessor can operate it is necessary for it to be able to access program instructions and data from either random access memory (RAM) or from read only memory (ROM). From a conceptual viewpoint we can regard this additional memory as a collection of 8-bit *memory locations*, each of which is identifiable by a specified unique *address*. With the 8-bit processors that we are using, we identify a particular memory location by specifying a 16-bit address so *two* bytes are required to specify a given location in the 0 - 64K range. Certain memory addressing instructions enable less than the full address to be given and this can have advantages in terms of speed of operation.

The descriptions that follow show, in a schematic form, the registers available in the Z-80 processor. The specific functions will be dealt with as we start to examine the instruction set and the facilities offered by the microprocessor. Since most processors have certain facilities and characteristics that are almost universal, it is useful to point these out while we look at our 'models' so that the overall general theme is apparent:

- All processors have a specialized register, called an *Accumulator*, used for arithmetic/Boolean functions.

- All processors have a *Program Counter* register that tells the microprocessor from which memory location the next instruction should be retrieved.

- It is necessary for a processor to be able to store items such as subroutine return addresses and the Z-80 processor uses a very common software method based on a selected area in RAM memory. This area is called a *Stack* because items are added to it and taken from it in the same way as you would for example, take cards from and add them to the top of a pack of playing cards (on a last in–first out principle). Most processors have a *Stack Pointer* register that determines the memory location that is the current 'top' of the stack. Instructions that place items onto the stack automatically adjust the stack pointer accordingly. The stack is also available for storage of other items such as the temporary saving of the microprocessors internal registers (through inbuilt instructions). It is common practice to talk of the placing of data on the stack as '*pushing*' onto the stack, when items are removed the terms '*popping*' or '*pulling*' are used.

- A selected set of bits within the processor are affected by certain conditions that can occur. These bits, called *Flags*, are frequently grouped together for storage on the stack etc. and are often collectively termed a '*status word*' or 'program status word'. As an example, any arithmetic operation that results in zero being present in the accumulator, will set the zero flag to 1 (remember a bit can only take values of 0 or 1 and, by convention, 1 is chosen to represent the 'true' condition).

- Eight-bit processors usually provide some means of combining certain of their 8-bit registers so that a 16-bit (i.e.two bytes) *memory address* can be specified.

- As well as the above registers a microprocessor will provide others, some for specific purposes such as indexed addressing and interrupt handling. Others, sometimes called 'secondary registers' are of a more general nature and can be used as desired.

LAYOUT OF THE Z-80 MICROPROCESSOR

The Z-80 processor is conceptually similar to an older microprocessor, the *Intel* 8080, but has much improved facilities. Twin sets of the main processing registers are available. Two 16-bit index registers are also present together with two specialized registers that will not concern us at present. A schematic description of the processor is shown in figure 4.1.

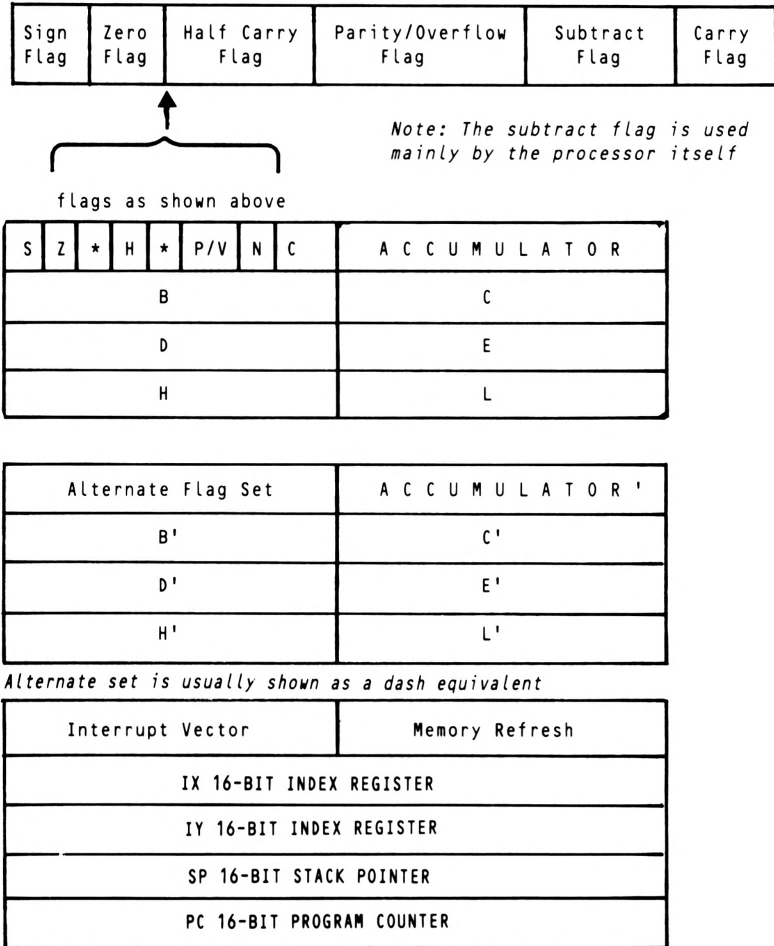


Figure 4.1: **Schematic layout of the Z-80 processor**

HEXADECIMAL NUMBERS

An 8-bit binary number can take values from 0 to 255 i.e. from 00000000_{bin} to 11111111_{bin}. It is often convenient to be able to express these values in a hexadecimal form which involves a '*base*' of 16 rather than the 'base 2' binary form. The use of different bases sometimes causes problems so before you say: 'I was never taught about bases at school', we'll tell you that you are probably already using different bases almost every day of your life, possibly without realizing it. If you still weigh things in pounds and ounces, then you are already working in a hexadecimal based number system – only the notation is different.

With our 'normal' numbering system, base 10, we group by units of ten. When you add numbers together you add the units first, then remove multiples of ten and 'carry' these over to a 'tens' column. You do a similar operation with the 'tens' column and so on. The only reason that 10 is used as the base or 'radix' of our normal numbering system is that it is found convenient: we are, after all, born with ten fingers to count on.

When you have to add 'pounds and ounces' you proceed as follows: You add the ounces and if they come to more than 16 you 'carry' the number of multiples of 16 into the 'pounds' column:

```
   2 lbs 14 ounces
+
   2 lbs 5 ounces        14 + 5 = 19: One group of 16 to carry
   ---------------                    and three units left over
   5 lbs 3 ounces
```

If instead of using numbers from 0 to 15 for ounces we used 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F then we could write 2lbs 14 ounces as 2E ounces. It is this *extended numbering* system that forms the basis of the hexadecimal notation.

The extension of our basic number symbols (the digits 0 to 9), could have been done by adding any extra symbols, but since we already have an alphabet, the use of letters to represent our extra numbers was an obvious extension. So in just the same way that you learnt what the numbers 0 to 9 symbolize it is necessary to learn that A_{hex} represents 10 in 'ordinary' base 10 numbers. In computing applications it is common to write 'H' or 'hex' after a hexadecimal number to avoid confusion with decimal numbers.

If you now consider the eight bits of a byte of binary information as two groups of four bits you will appreciate that we can represent each of those two groups by a hexadecimal digit. Consider the binary number 10001111:

```
1000 Binary = 8 decimal = 8 hex
1111 Binary = 15 decimal = F hex
```

Thus 1000 1111 binary can be written very compactly as $8F_{hex}$. In a similar fashion we can represent two bytes (i.e. 16 bits of information), by using *four* hexadecimal digits.

1111 0000 1000 1111 binary can be split into four groups of four bits and written in hex form as $F08F_{hex}$. For specifying addresses – locations in memory – you will find the hexadecimal notation *very* convenient.

PAGES OF MEMORY

We have already said that it is possible, with a two byte address, to specify any one memory location out of a total of 64K (i.e. 64 x 1024 = 65536) such locations. Such an address can be written in hex form using four hexadecimal digits. An additional concept of dividing available memory into *pages*, each of 256 locations, has also proved useful. The first byte or 'high byte' of such an address is therefore often called the 'page number'. *Page zero* then refers to the initial 256 bytes of memory, whose addresses go from 0000_{hex} to $00FF_{hex}$.

THE Z-80 INSTRUCTION TYPES

There is no universal standard by which to group the various instructions that a microprocessor can execute. The following classification is a simplified version of the details commonly available from manufacturers and other sources.

Data transfer instructions

On the Z-80 these fall into various categories: 8-bit, 16-bit, block transfers and operations using the Z-80's stack.

8-bit transfers

These are accomplished by the *load* instructions. These instructions are written in Z-80 assembly language as LD destination, source.

For example, to load the accumulator (designated as register A) from register C we use the instruction LD A, C.

Such transfers can be made between any two working registers: A, B, C, D, E, H or L. No direct instructions exist that enable values to be exchanged between the 'active' and the 'alternate' register set; when required, such transfers may be achieved by use of the stack or additional memory.

In order to load any register other than the accumulator from a memory location it is necessary to place the location address into the HL register pair. HL thus points to the location to be used as the source. The notation (HL) signifies that it is the *contents* of the byte addressed by HL that is involved in the transfer. Thus LD C, (HL) will load the C register with the contents of the byte whose address is contained in HL.

Similarly to store the contents of register C at the location whose address in in HL we would use the instruction LD (HL), C.

The accumulator is the only register that can store data directly to, and load data directly from, a memory location. LD A, (0FFFH) will

load the accumulator with the contents of the byte whose address is $0FFF_{hex}$. Similarly LD (0FFFH), A will transfer the contents of the accumulator directly to the specified memory location.

Data can also be loaded from a byte that follows the op code byte. eg. LD B, 0FH will load the B register with the value 15, i.e. 0F hex. This is termed immediate addressing and is explained later.

Other possibilities including loading using indexed addressing are available and the instruction set listing covers these in detail.

16-bit transfers

Any of the register pairs BC, DE, HL, SP, IX or IY may be stored at, or loaded from, a specified pair of memory locations. They may also be loaded with a 16-bit value contained in the two bytes following the instruction. Additionally BC, DE, HL, IX, or IY may also be transferred to, or loaded from, the top of the stack (whose address is kept in the stack pointer or SP register). The accumulator and the flag status are also treated as a 16-bit register that can be transferred to and retrieved from the stack. Miscellaneous other instructions exist that perform 16-bit transfers.

EXAMPLES

PUSH DE
This decreases the stack pointer then pushes the contents of the D register onto the stack. The stack pointer is again decreased before the contents of the E register are placed on the stack – note that the stack is growing downward in memory as items are added.

POP DE
This performs the reverse operations to the PUSH instruction described above.

LD BC, 10FCH
This loads the BC register pair with the 16-bit value $10FC_{hex}$.

Block transfer instructions

The Z-80 has some quite powerful, and extremely useful, block transfer instructions. All of these instructions use three sets of registers namely BC, DE and HL. The BC pair are used as a 16-bit counter which specifies the number of bytes involved in the transfer. HL is used to point to the source, and DE used to point to the destination. The instructions have the mnemonics LDD, LDDR, LDI, LDIR and their operation is explained in the instruction set listing.

Data processing instructions

Various addition and subtraction instructions exist, both with and without carry. ADD A, B for example will add the contents of the B register to the accumulator. The instruction ADC A, B will also add the contents of the B register to the accumulator but if the carry flag is set this will also be added into the result. In both cases the result is stored into the accumulator and thus overwrites the original accumulator contents.

Several logical instructions, AND, OR and XOR (eXclusive OR) are available together with instructions that enable 8-bit data values to be compared, rotated and shifted in various ways. Details are provided within the instruction set listing and examples of their use will be found in the text.

Conditional test instructions

Many of the instructions available on the Z-80 will use particular flag conditions as a criteria for execution. These include the *conditional jump*, *conditional branch* and the *conditional call* instructions. As an example the instruction CALL Z, INPUT$ROUTINE will call a subroutine whose address is defined by the label INPUT$ROUTINE only if the zero flag is set. If the zero flag is not set then the subroutine will not be called.

Miscellaneous instructions

Numerous other instructions exist that are used for *input and output*, *interrupt* servicing, forcing particular *flag* conditions, switching between *alternative register sets*, and so on. Certain instructions have been deliberately excluded from our simplified listing and in such cases you are referred to the manufacturers data or other sources.

No Z-80 book would be complete without a mention of the extensive '*bit manipulation*' instructions that are available. The three instructions BIT, RES (RESet bit), and SET enable individual bits of a register, an indirectly addressed byte, or an indexed addressed byte to be tested, reset or set as required. We have avoided using such instructions because they are not generally available on other microprocessors; you should, however, be aware that they do exist and are often useful.

EXAMPLES

BIT 4, A
Test bit 4 of the accumulator and set zero flag accordingly.

SET 2, (IX+d)
Set bit 2 of the indexed addressed byte (IX+d).

RES 5, (HL)
Reset bit 5 of the indirectly addressed byte (HL).

We have not attempted to cover the uses of all of the Z-80 instructions that exist. The main area not covered relates to those instructions used in operating system programming, these include the use of interrupts (which are essentially hardware generated subroutine calls) and the port input/output instructions. Such material is not particularly relevant for our purposes and in the early stages can be quite easily remain 'hidden' behind the operating system that your computer uses.

INSTRUCTION FORMAT

Instructions on the Z-80 can consist of between one and four bytes. In general the instructions with the shortest length execute the most quickly.

Single byte instructions

One example is the load instruction LD A, C. As mentioned above this will load the contents of the C register into the accumulator. When assembled, a single byte is produced, which has the value 79_{hex}. This byte of '*object code*' that is produced is the numerical equivalent of this particular assembly language instruction. It is called the *op-code* for the instruction.

Two byte instructions

A typical example is the instruction LD C, 0FH which will load the C register with the value $0F_{hex}$. When assembled two bytes of object code are produced. The first byte is $0E_{hex}$, which is the op-code for the instruction, the second byte contains the value 0F hex, the value specified in the instruction.

Three byte instructions

The 16-bit equivalent of the load instruction shown above is LD BC, 1F20H which loads the BC pair with the 16-bit value shown. Assembly of this instruction produces one op-code byte followed by two bytes that hold the value $1F20_{hex}$. Other examples include instructions that involve specifying 16-bit addresses, eg. LD A, (0F21H) which is assembled to produce the op-code byte, followed by the low order part of the address (21_{hex}) followed by the high order part ($0F_{hex}$).

Four byte instructions

Some instructions, namely those involving the indexed registers or indexing, can require four bytes. One example is the instruction LD (IX+d), n which loads the byte whose address is found from the contents of the index register IX plus a displacement 'd' with the value specified by the letter n. Once assembled the instruction produces the following object code bytes:

- A two byte op-code, DD_{hex} and 36_{hex}
- A single byte containing the value of the displacement
- The specified data value 'n'

THE Z-80 INSTRUCTION SET

Appendix A lists the Z-80 instruction set in mnemonic form, giving a description of each operation together with details of its effect on the flags, a listing of the object codes generated and miscellaneous other information.

5 SEQUENCE and REPETITION

One of the observations that led to the concept of structured programming was the discovery that virtually all problems may be solved by using a combination of the three basic structures: *Sequence*, *Repetition* and *Alternation*.

The term *sequence* simply means doing things in a serial order. We can illustrate this in a flowchart form as shown in figure 5.1:

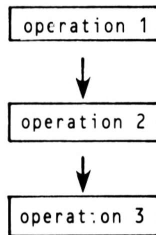


Figure 5.1: **Sequence in flowchart form**

The equivalent Warnier diagram is shown in figure 5.2:

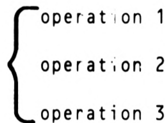


Figure 5.2: **Sequence in Warnier diagram form**

Repetition concerns itself with performing a set of actions a given number of times. Program *loops* such as BASIC's 'FOR .. NEXT', 'DO .. UNTIL' and 'WHILE .. WEND' are typical examples of this type of structure. Two forms are possible, depending on whether we are testing the exit condition *before* (pre-test) or *after* (post-test) performing the required processing. Figures 5.3 and 5.4 show how we can use flowchart and Warnier representations to illustrate repetition.

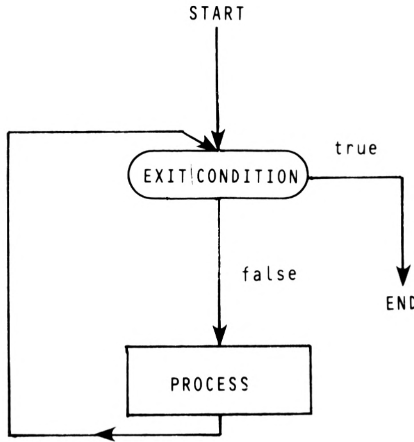


Figure 5.3: **Pre-test Repetition Flowchart form**

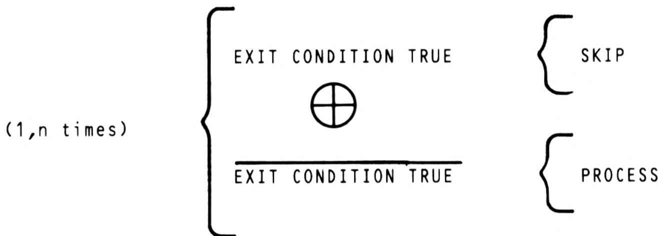


Figure 5.4: **Pre-test Repetition Warnier diagram form**

The description of alternation will be left until the next chapter when we shall examine it in detail.

REPETITION – An example program

As an illustration of repetition, we have chosen a simple program to collect characters directly from the keyboard, and to print them on the VDU screen. There are many ways such a program could be written using BASIC. We have chosen one particular representation that will let us compare the assembly language equivalents very easily.

The BASIC program can be divided into three parts. An initial or 'Setup' block, whose main task is to define a variable called CARRIAGE.RETURN$ as the string equivalent of the ASCII 13 'Carriage Return' code. An 'end' block, which in our example is nothing more than a single END BASIC statement. The bulk of the program is labelled the 'main' block and performs the following functions: We collect a character, using the INPUT$() function; we

then check to see if it is a carriage return character, if it is we jump to the end of the program, otherwise we print the character and jump back to the input statement in line 30 for the next character. Here is the BASIC form:

```
* ===================================================================
SETUP     10   CLEAR
BLOCK     20   CARRIAGE.RETURN$=CHR$(13)

          30   X$=INPUT$(1)                      'collect character in X$
MAIN      40   IF X$=CARRIAGE.RETURN$ THEN 70 'end of input if true
BLOCK     50   PRINT X$;:                        'output character to VDU
          60   GOTO 30                           'get next character

END       70   END
BLOCK
* ===================================================================
```

When we write an equivalent program in assembly language we use the same type of program structure. The 'Setup' block will, however, vary according to your assembler, and your operating system. We give a typical example of the type of coding that could be expected.

Z-80 mnemonics for SETUP BLOCK

```
CARRIAGE$RETURN      EQU  13
                     ORG  100H
                     JP   STACK
                     ORG  150H
STACK:               LD   SP,$-2
```

OPERATIONS USED

- Define CARRIAGE$RETURN so that the assembler will recognize this term as meaning the number 13.
- Define whereabouts in memory your program is to start. Our assemblers use ORG (short for ORiGin) and in the above example the program starts at 100_{hex}.
- We perform an *unconditional jump* to an address that is labelled STACK. This is the first real assembly language instruction we have encountered. This type of jump is called an *unconditional* jump because it is performed irrespective of any processor flag conditions. The mnemonic JP represents the start of a three byte instruction. The first byte is the 'op code', i.e. the numerical representation of the mnemonic, the second and third bytes are

the required jump address. A schematic description is shown in figure 5.5.

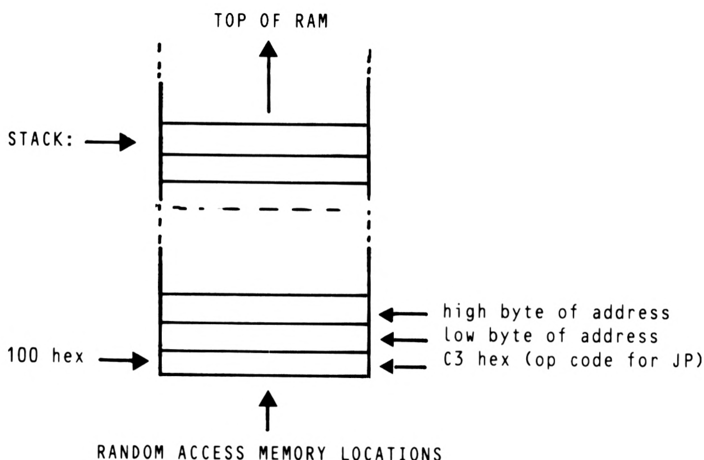


***Figure 5.5:* Schematic representation of a 'Jump' instruction**

- The microprocessor performs this jump by placing the address following the op code into the program counter register. The program counter is called the destination register for the information transfer. The information source is the two bytes that immediately follow the op code.

  In general the term 'addressing' refers to the specification, within an instruction, of the location of the 'operand' – the byte(s) upon which the instruction will operate.

  Since the operand for the JP instruction is the two immediate data bytes that follow the op code the addressing mode being used to execute the instruction JP is called 'immediate'.

- Next we tell the assembler to 'move over' the space we are going to reserve for our stack. This is done by using ORG 150H which results in the assembler placing our next mnemonic instruction at this new origin, thus reserving half a page of memory for the stack.

- Lastly we load the stack pointer register with the value `$-2`. Our assemblers use '`$`' to define the address of the current memory location. The Z-80 instruction that loads the stack pointer register is called a *load* instruction. The mnemonic is LD and this instruction can be used to load a 16-bit address into either the BC, DE,or HL register pairs or to load a 16-bit address into the stack pointer register. The letters SP indicate that we are loading the stack pointer register. So we are loading the stack pointer just below the address of the memory location called `STACK`. Remember that we do not need to know the numerical address of the location that we have labelled `STACK` – we leave that to the assembler. Bear in mind that frequently it is possible to use a mnemonic with a variety of addressing modes. We have used the LD instruction in an 'immediate addressing mode' but we will see later that other uses are also possible. You will be better able to appreciate this after we have defined and examined addressing techniques.

Remember also, that EQU and ORG are assembler 'pseudo operations', not mnemonics. They are not assembled into program statements. They simply tell your assembler that you wish to define a term, or define the start of a program. You may find that your system uses different conventions. It may use START instead of ORG for the starting address. You will need to find the address that your system expects your programs to start at. Similarly the EQUate 'pseudo op' may be expected in a different form such as CARRIAGE$RETURN:=13, but these things are dependent on your system and your assembler – *you* must check your manuals to ensure that you are complying with the syntax and other requirements.

We have indicated the general type of setup block usually required. It may be that your particular system requires joint use of the stack and that your programs should simply use an existing stack. In other cases it is necessary to save the operating system's stack pointer so that it can be re-instated when your program has finished. You must, to a large extent, be guided by your own system requirements.

We can look now at the coding of the main block. We are going to show two forms so that we can illustrate the differences between some of the jump instructions that the Z-80 can perform. We have seen an unconditional jump in the setup block. Now we get the chance to see a jump instruction which is conditional on the state of one of the Z-80's flag bits. As you look at the examples compare them with the BASIC form given earlier.

Z-80 mnemonics for MAIN BLOCK code

```
START:    CALL INPUT$ROUTINE      ;collect character in accumulator
          CP   CARRIAGE$RETURN    ;end of input if true
          JP   Z,FINISH           ;<-- conditional jump
          CALL OUTPUT$ROUTINE     ;output character to VDU
          JP   START              ;<-- unconditional jump
```

Alternative MAIN BLOCK

```
START:    CALL INPUT$ROUTINE      ;collect character in accumulator
          CP   CARRIAGE$RETURN    ;end of input if true
          JR   Z,FINISH           ;<-- conditional relative jump
          CALL OUTPUT$ROUTINE     ;output character to VDU
          JP   START              ;<-- unconditional jump
```

Firstly we call a subroutine that we have called INPUT$ROUTINE to collect a character in the accumulator. The Z-80 mnemonic for a subroutine call is CALL. This instruction places the address `INPUT$ROUTINE` into the processor's program counter so that a jump to the required subroutine occurs. Prior to this the program counter, which is pointing to the next instruction, is automatically pushed on the stack. When the subroutine terminates, the address that was stored on the stack is 'popped' off and placed back in the program counter. The program counter is then pointing to the instruction after the CALL instruction. In this way the processor can 'remember' where it should return to after the subroutine has been performed.

The next instructions compare the character collected in the accumulator with the value `CARRIAGE$RETURN`. The processor subtracts from the contents of the accumulator the value of the byte specified (in this case 13). If the contents are equal then the result of the subtraction is zero and the processor's zero flag will be set. The result of the subtraction is not stored anywhere and the contents of the accumulator are not altered. Only the processor status word, i.e. the flags, are affected.

Immediately following the comparison test we have placed in the first example an instruction called a conditional jump. If the zero flag has been set then a jump occurs to an as yet unspecified FINISH routine. The second example uses an alternative instruction known as a *conditional relative jump*.

If the zero flag has not been set then the jump to `FINISH` does not occur, so a further subroutine call, this time to an output routine, is made. We then jump back to the start of the Main block to collect another character.

It is necessary here to point out that there is a distinct difference between the JP instruction of the Z-80 processor, and the JR instruction. The former instruction results in a jump to an address that has been specified by a two-byte operand. The JR instruction is a 'relative conditional jump' instruction, and this is using a different form of addressing known as 'relative addressing'. The value of the operand is a one-byte displacement, not an address. The relative jump or 'branch' as it is sometimes called is limited to values that can be specified within one byte – your assembler will calculate the displacement, and should tell you if you exceed this limit.

Relative addressing has the advantage of only requiring a two-byte instruction (which makes for faster execution). Since we do not use an absolute address it also means that the code produced is 'relocatable'.

The disadvantage is that you are limited to displacements that can be specified with one byte. The allowed displacement +127 to –128 gets added to the contents of the program counter. The full explanation of this instruction must wait until we have examined two's complement arithmetic.

To finish your program you are again in the hands of those who designed your system. Your manuals will tell you how your program may finish. Programs operating in a CP/M environment can use a JP 0 instruction to 'reboot' the operating system. In such a case FINISH would consist of the following line:

```
FINISH:    JP  0     ;Reboot operating system
```

INPUT AND OUTPUT routine

Once again you are very dependent on your operating system, you will find memory addresses for functions like *direct input* and *console output*. These might be given abbreviation names such as GETCHAR, CONIN and OUCH or CONOUT. They enable you to avoid the complexities of input and output by using existing operating system routines.

If your operating system allows you to call these functions directly then all that needs to be done is to place additional EQUate definitions into your setup block. This is quite satisfactory in our example because we are only using the accumulator register, but frequently the operating system when performing these functions will affect some or all of the microprocessor's other working registers. In general it is normal practice to save the contents of the internal registers by '*pushing*' them onto the stack before using an operating system function.

One problem that may occur is that the direct input function does not wait for input; you may find that your manual says that if a

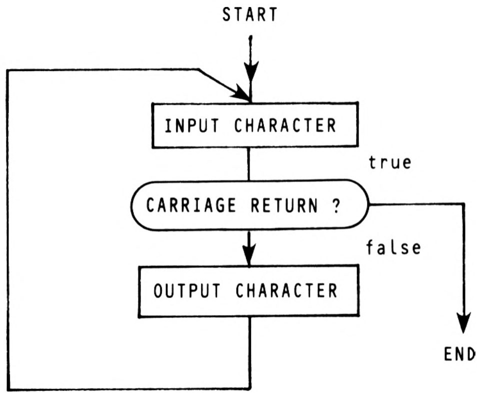


Figure 5.6: **Flowchart for the example program**

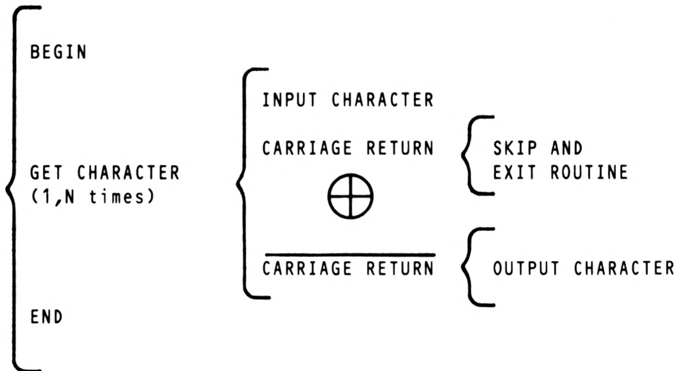


Figure 5.7: **Warnier diagram for the example program**

keyboard character is not available the operating system routine will return with the value *zero* in the accumulator. If this is so then you must create a 'wait for input' loop. You can do this by using some of the instructions we have already looked at. The idea is fairly straightforward and you have already seen most of the instructions needed:

```
INPUT$ROUTINE: CALL SYSTEM$INPUT$ROUTINE   ;system direct input
               CP   0                      ;is accumulator zero
               JP   Z,INPUT$ROUTINE        ;no input so keep waiting
               RET                         ;return from subroutine
```

As long as the system function is returning with zero in the accumulator then the zero flag is being set by the compare instruction, thus we keep looping back to the start of the input routine until a character is collected.

In this case you would use an equate operation in the setup block to define SYSTEM$INPUT$ROUTINE – the address of the system input routine. You would also have to place the above subroutine into your program. The only instruction that is new in the above subroutine is the RET instruction. You remember we said earlier that, when you call a subroutine, the address of the next instruction is pushed onto the stack; well, by making the last instruction of a subroutine a RET then that address is 'popped off' the stack and placed back in the program counter register. The next instruction to be executed after the completion of the subroutine is then the one that followed the original subroutine call instruction.

PUTTING IT ALL TOGETHER

Having explained how our example program operates we show you the flowchart and Warnier forms of the program (figures 5.6 and 5.7) – and the outputs from our assemblers – so that you can see what the complete program looks like in its assembly language form.

```
* ===================================================================
*              CHAPTER 5 EXAMPLE..............CP/M VERSION
* -------------------------------------------------------------------
* CP/M Version Notes: This operating system requires that you
* identify the system function needed by placing a "function
* number" into the microprocessors C register. It also expects
* "output characters" to be in the E register and not the
* accumulator. This means we have to use instructions to transfer
* the contents of the accumulator to the E register. We set up the
* necessary details and then CALL the operating system through a
* common entry point which is a jump located at memory location
* 05 hex. The direct I/O function used also needs FF hex
* the E register to indicate that input (rather than output) is
* required.
```

```
* ==================================================================
*                       S E T U P - B L O C K
* ------------------------------------------------------------------
CARRIAGE$RETURN            EQU     13
OPERATING$SYSTEM           EQU     5
                           ORG     100H
                           JP      STACK
                           ORG     150H
STACK:                     LD      SP,$-2
* ==================================================================
*                       M A I N - B L O C K
* ------------------------------------------------------------------
START:                     CALL    INPUT$ROUTINE
                           CP      CARRIAGE$RETURN
                           JP      Z,FINISH
                           CALL    OUTPUT$ROUTINE
                           JP      START
* ==================================================================
*                       E N D - B L O C K
* ------------------------------------------------------------------
FINISH:                    JP      0       ;Reboot operating system
* ==================================================================
*                   I N P U T - R O U T I N E
* ------------------------------------------------------------------
* Notes:  We have to use a "wait for input" loop here. With this
* system  (CP/M)  it is necessary to preserve the contents  of  the
* registers before using the operating system calls.
* ------------------------------------------------------------------
INPUT$ROUTINE:           PUSH BC ! PUSH DE ! PUSH HL  ;Preserve reg's
INPUT$ROUTINE$1:         LD     E,0FFH      ;Signifies console input
                         LD     C,6        ;Direct cons I/O function
                         CALL   OPERATING$SYSTEM
                         CP     0                   ;0 = no key pressed
                         JP     Z,INPUT$ROUTINE$1
                         POP HL ! POP DE ! POP BC  ;Restore registers
                         RET
* ==================================================================
*             O U T P U T - R O U T I N E
* ------------------------------------------------------------------
OUTPUT$ROUTINE:          PUSH AF ! PUSH BC ! PUSH DE ! PUSH HL
                         LD     E,A        ;Transfer is in E register
                         LD     C,2          ;Console output function
                         CALL   OPERATING$SYSTEM
                         POP HL ! POP DE ! POP BC ! POP AF
                         RET
* ==================================================================
```

6
ALTERNATION

We defined *sequence* and *repetition* in the last chapter and illustrated the ideas with a short program. Now it's the turn of 'alternation', the third and last of our structured building blocks. Simple alternation is exemplified by BASIC's 'IF .. THEN .. ELSE' type of coding. The essential features can be illustrated using flowchart and Warnier diagram illustrations as shown in Figures 6.1 and 6.2

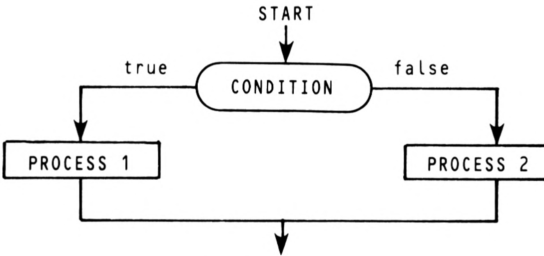


***Figure 6.1:* Simple Alternation in Flowchart form**

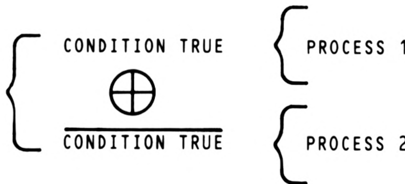


***Figure 6.2:* Simple Alternation in Warnier diagram form**

This form is called '*simple*' alternation in order to distinguish it from those cases that involve more than two alternatives. We are implying, in both representations, that any necessary preselection processing will have been performed.

A choice is being made between two sets of actions based on a specified condition. Simple or '*binary*' alternation represents the existence of two mutually exclusive operation subsets. The ideas can be generalized to condition tests with N mutually exclusive outcomes. This leads to the corresponding existence of N mutually exclusive operation subsets within the logical program description.

We want to give you an illustration of how we can create 'alternation constructs' when writing programs in assembly language. To keep to familiar ground we shall examine a slightly more complex problem related to the 'collection of characters' program of chapter 5. In this way the flowchart and Warnier forms will be less intimidating and the assembly language programs will have a certain amount of material that will be familiar already.

The same approach is used: we will look at the problem, consider the principles involved in flowchart and Warnier diagram form, then give a typical BASIC solution followed by the assembly language forms and their explanations.

A SAMPLE PROBLEM

We want to write a routine that will collect input from a keyboard and differentiate between control characters and printable characters. The routine should terminate when the *carriage return* (ENTER or RETURN on most computers) key is pressed. Other control characters less than ASCII value 32 are to be ignored although a warning *bleep* is to be given. All other input characters should be echoed to the VDU screen.

AND ITS SOLUTION

The problem is straightforward and is well defined. We need some sort of input routine; we need to compare each character that is collected to see if it is a carriage return (ie. has an ASCII value of 13). If not we need to know if it is another control character or a character to be printed. Figure 6.3 shows the flowchart representation:

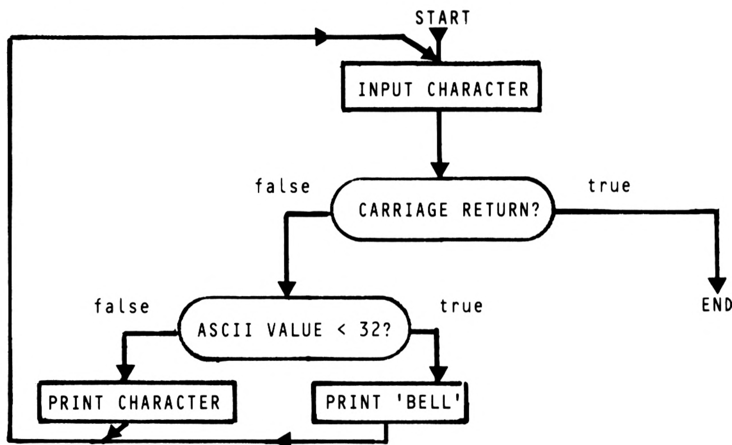


Figure 6.3: **Flowchart for the example program**

Look at the equivalent Warnier diagram representation in figure 6.4:

What does figure 6.4 tell us? We collect a character using an input routine. If the input character *is* a carriage return then we exit from the routine. If it is *not* a carriage return then we make a further test to identify whether the character is a control character or one to be printed. Having performed one of two possible alternative sets of actions we return for another input character.

How would we program this in, say, BASIC? Well, let us first look at one translation from the Warnier diagram in a Microsoft BASIC form:

```
10 X$="A"              ' must force an entry into WHILE/WEND loop
20 WHILE ASC(X$)-13    ' <>0
30 GOSUB 1000          ' some input routine collects input in X$
40 IF ASC(X$)<32 THEN PRINT CHR$(7) ELSE PRINT X$
50 WEND
60 END
```

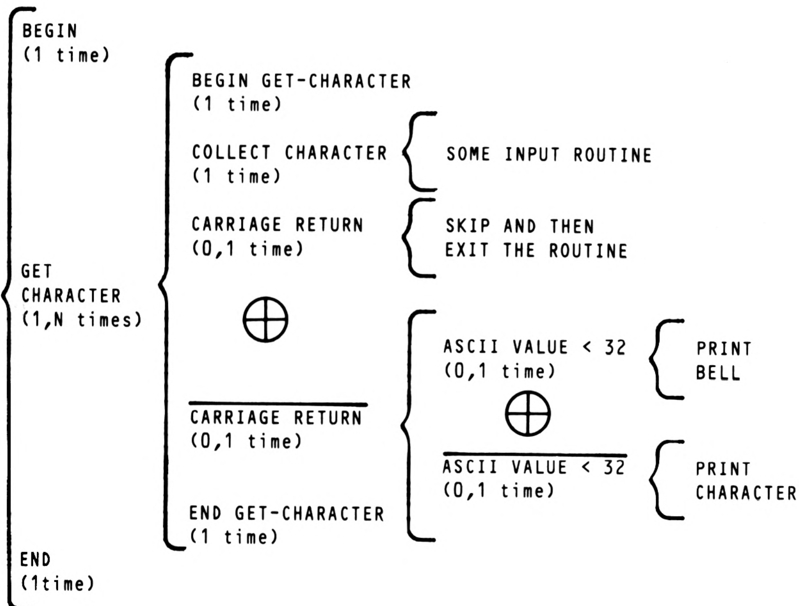


Figure 6.4: **Warnier diagram for the example program**

In our problem we can print 'bell' and other characters directly; but in general you may want to perform more than just a single instruction as the result of a conditional test. This being so a more generalized equivalent is to be preferred:

```
10 X$="A'               ' must force an entry into WHILE/WEND loop
20 WHILE ASC(X$)-13     ' <>0
30 GOSUB 1000           ' some input routine collects input in X$
40 IF ASC(X$)<32 THEN GOSUB 2000 ELSE GOSUB 3000
50 WEND
60 END
```

Subroutine 2000 would perform those actions concerned with 'printing a bell' and subroutine 3000 would concern itself with printing a character.

Yet another equivalent form, and one that in terms of coding is arguably more efficient, can be obtained by using the 'dreaded GOTO':

```
10 GOSUB 1000        ' some input routine collects input in X$
20 IF ASC(X$)=13 THEN END
30 IF ASC(X$)<32 THEN GOSUB 2000 ELSE GOSUB 3000
40 GOTO 10
```

Such a form is perfectly acceptable and shows a correct use of `GOTO`. The arguments against such code stems, not from the proper use of the `GOTO` statement, but from the fact that they can easily be used incorrectly. When they are used incorrectly they create tangled code which is difficult to maintain, difficult to understand and prone to errors.

We can make a point in passing that when one talks of a 'structured language', one is usually suggesting that the available constructs of the language will attempt to force a user into writing in a particular way. The object is to avoid giving the user the chance to create problems by incorrectly use of things like unconditional jumps.

- You should not be misled into thinking that, because a language is labelled 'unstructured', it is not possible to write well structured programs using that language.

The Carry flag

We have already used 'immediate' comparison instructions to test for equality. The instruction used was the CP operand. When the contents of the accumulator are the same as the immediate byte specified, then the internal subtraction that occurs during the comparison, results in the zero flag being set.

When the above comparison instructions are used, several flags are affected. Our present concern is the effect of these operations on the 'carry' flag.

- If the value being compared is greater than the value present in the accumulator then the microprocessor's carry flag will be set.

We can tabulate all possible outcomes of such testing in the following manner:

Condition	Carry flag	Zero flag
Accumulator > Immediate Byte	Cleared	Cleared
Accumulator = Immediate Byte	Cleared	Set
Accumulator < Immediate Byte	Set	Cleared

In all cases the contents of the accumulator, and of the immediate byte value specified, are treated as simple binary data.

We are going to use the carry flag to detect control characters – characters having an ASCII code of less than 32.

Let us look at the main part of a Z-80 assembly language interpretation of the above forms and make some observations:

```
* ==================================================================
*                  Z 8 0 - V E R S I O N - 1
* ------------------------------------------------------------------
START:                  CALL INPUT$ROUTINE    ;Character in accumulator
                        CP   CARRIAGE$RETURN  ;End of input if true
                        JP   Z,FINISH
                        CALL NOT$CARRIAGE$RETURN
                        JP   START           ;Loop back for next character
* ------------------------------------------------------------------
NOT$CARRIAGE$RETURN: CP   SPACE
                        CALL C,  CONTROL$CHARACTER
                        CALL NC, PRINTABLE$CHARACTER
                        RET
* ==================================================================
```

The Z-80 mnemonics CALL C and CALL NC stand for 'call on carry' and 'call on no carry' respectively. They illustrate the concept of a conditional subroutine call. The function of these instructions is to perform the specified subroutine call – but only if the necessary flag condition is satisfied.

A more efficient form of coding is shown below. It is more compact and satisfies the requirements of our problem but you will see later that you can sometimes run into problems which, at present, are not immediately obvious.

```
* ===================================================================
*                 Z 8 0 - V E R S I O N - 2
* -------------------------------------------------------------------
START:            CALL INPUT$ROUTINE      ;Character in accumulator
                  CP   CARRIAGE$RETURN       ;End of input if true
                  JP   Z,FINISH
                  CP   SPACE
                  CALL C,  CONTROL$CHARACTER
                  CALL NC, PRINTABLE$CHARACTER
                  JP   START         ;Loop back for next character
* ===================================================================
```

We use an input routine to collect a character. This is returned in the accumulator register. The CP instruction compares the value of CARRIAGE$RETURN (which will have been previously set to 13 by an EQU directive) to the ASCII value of the character present in the accumulator. If the character present in the accumulator is a carriage return the zero flag will be set. As in the first form, the JP Z instruction following this means we exit from the routine as soon as a carriage return character is detected.

If the character being looked at is not a carriage return, then we compare the accumulator contents to the value SPACE (again previously defined by using an EQU directive). If the character present in the accumulator has an ASCII value less than 32, then the Z-80's carry flag will be set. Otherwise the carry flag will be clear.

In both of these examples we are using the carry flag to implement the equivalent of an 'IF .. THEN .. ELSE' structure. If the carry flag is set we do one set of operations. If the carry flag is not set we perform the alternative set of operations. The only stipulation that has to be made is that the status of the flag being tested must be preserved by the first of the subroutines that may be called.

CONTROL CHARACTER SUBROUTINE

This has to output a bell character. On most terminals this is done by sending the ASCII bell character to the terminal. In BASIC you use 'PRINT CHR$(7)', in assembler you load a register with the value 7 and then use your system output routine to send the character to the terminal. The normal procedure is to define BELL using an EQUate pseudo operation and the example shown below assumes that this has been done.

The Z-80 has instructions to load a specified register with an 8-bit data value. The form the instruction takes is:

LD register, 8 bit data value

```
* ==================================================================
*                 Z 8 0 - V E R S I O N
* ------------------------------------------------------------------
CONTROL$CHARACTER:  LD   A,BELL
                    CALL OUTPUT$ROUTINE
                    RET
* ==================================================================
```

The LD mnemonic is however, also used to represent register loading operations other than the loading of immediate data values. LD on the Z-80 processor, when used as shown above, is using immediate addressing; additionally it is used to represent data transfer using other addressing modes.

Go back now and look at the flowchart we are using for the example program. Can you pick out the subset of actions associated with '*not finding a carriage return character*'? You will probably agree that even in this simple example the isolation of such subsets are not particularly obvious.

Try to find the same subset on the Warnier diagram, remember that we write the logical opposite of a statement by placing a bar over the statement. The subset we are discussing is shown in figure 6.5 below:

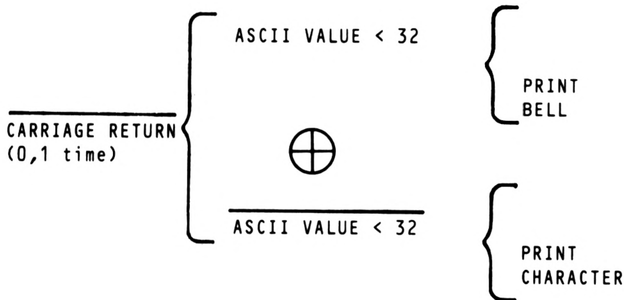


***Figure 6.5:* An isolated subset of actions**

The reason we are interested in this subset is that the first version explicitly treated the coding involved as a distinct subset, that is to say that actions corresponding to '*not carriage return*' were implemented as a 'called subroutine'. The code is therefore related to the design diagram on this basis – the action subset is defined by coding as a subroutine. The advantage being that the 'structure' of the diagram and the coding is (dare we say it) isomorphic! (This is a word used by mathematicians to imply structural similarity.)

The coding in the second versions performs the same function as the coding in the first forms, but the action subset '*not carriage return*' that we are discussing is not explicitly defined in the second form

code. The difference may not be immediately apparent to you, so let's briefly digress to explain this point . . .

There is a real advantage, especially when writing large assembly language programs, in being able to easily locate the section of code that is relative to a particular action subset in the corresponding design diagrams. Such advantage is paid for by a slightly increased program size.

Hardcore assembly language programmers often take great exception to 'wastage of bytes' in this manner and in certain applications their objections are justifiable. Our defence in general terms is two-fold:

- It is often of great practical advantage to have coding that is isomorphic with the design diagrams.
- Memory is getting cheaper, debugging time is not! Anything that improves program 'readability' and 'maintainability' is very welcome, especially when it comes to assembly language programming.

Explicit subset definition based on isomorphism between the design diagram and the actual program code contributes in practice to significantly reduced debugging time. The message is simple. Save bytes by all means but distinguish carefully between pointless inefficiency and the deliberate choice of using a few more bytes to create code that can easily be compared to the design diagrams.

THE PRACTICAL SOLUTION

We have used our example to explain some general ideas. There is a very good reason why you would not, in practice, actually need to write subroutine-based code for this particular example. Look back at some of the coding we have given and think about how we 'output' printable characters, and how we 'output' the ASCII 'bell' character. In practice we shall be using the accumulator to output the printable characters, we will also use the accumulator to output the 'bell' character. We will also, in both cases, be using our system OUTPUT$ROUTINE to send the character to the terminal.

Let us now consider and modify our flowchart/design diagrams in the light of the above information. Figure 6.6 shows the revised flowchart representation.

When we consider fully the practical implementation of our problem we see that one of the alternation subsets is in fact a 'do nothing' process. The Warnier diagram equivalent is shown in figure 6.7 for comparison.

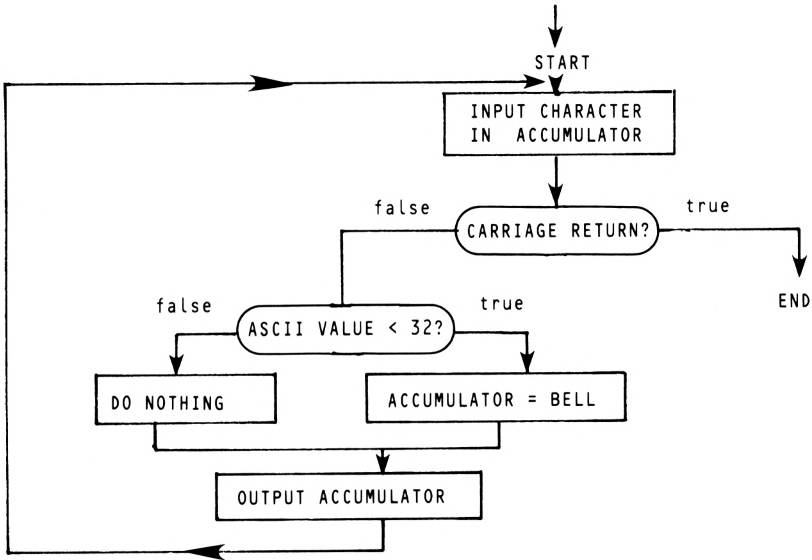


***Figure 6.6:* Modified flowchart for the example program**

This type of structure is frequently handled by simple 'in-line' conditional relative branching or conditional jumping. Based on a certain condition we either perform some section of code or we avoid it by jumping over it. Please bear in mind that this type of structure is a subclass of the simple alternation that we dealt with first. There are still, from a theoretical viewpoint, two sets of actions. The distinction is that one of the subsets is an 'empty set'.

Having possibly struggled through some the ideas we have presented so far you will no doubt be pleased to see the assembly language code that results from our most recent efforts.

If you have persevered up to this point you should find the code fairly straightforward. The label PRINT$CHARACTER identifies the location to be jumped or branched to if the carry flag is not set.

```
* ================================================================
*         Z 8 0 - P R A C T I C A L - S O L U T I O N
* ---------------------------------------------------------------
START:                CALL INPUT$ROUTINE
                      CP   CARRIAGE$RETURN
                      JP   Z,FINISH
                      CP   SPACE
                      JP   NC,PRINT$CHARACTER
                      LD   A,BELL
PRINT$CHARACTER:      CALL OUTPUT$CHARACTER
                      JP   START
* ================================================================
```

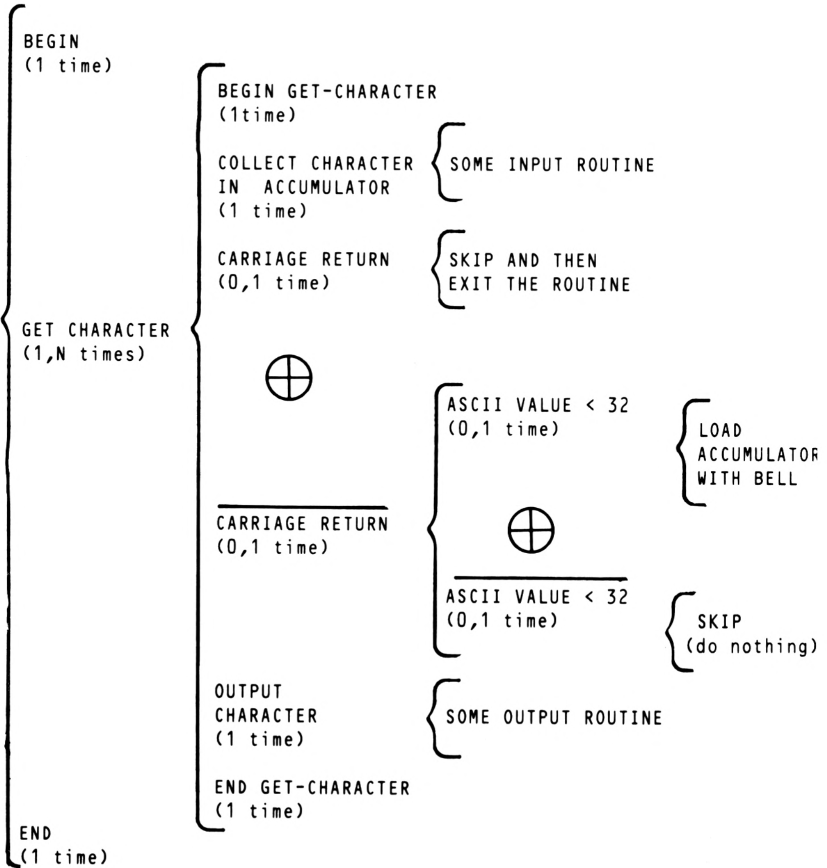


***Figure 6.7:* Modified Warnier diagram for the example program**

You should by now, appreciate the relationship between simple alternation, where two subsets of actions are involved, and the specific case of simple alternation where one of those subsets is an empty set. You have also seen some of the ways in which we can write the corresponding code.

It is important to grasp the general ideas involved because alternation – together with sequence and repetition – will occur in the majority of the programming problems that you will encounter.

You should appreciate that more complex alternation with multiple mutually exclusive action subsets can be described by extending the same basic principles that we have discussed.

FINAL WORD

One final point we wish to make now is that the design of our solutions is derived from the logical examination of the problem.

- The logical solution exists as an independent entity and by having such solutions available *before* you start coding you will side-step many problems that other approaches walk straight into.

By using this 'design before coding' approach, we find that we are left only with the much smaller problem of how to use an available instruction set to implement a logical solution that is already known. We don't ask you to accept this philosophy without question, but we would like you to think about the implications (and in particular the benefits) of having language independent solutions available before you start coding.

If you modify the main block in chapter 5 to incorporate the practical solution you should be able to run a version of the program. You might also like to experiment with some of the other ideas we considered.

```
* ==================================================================
*          CHAPTER 6 EXAMPLE............... CP/M VERSION
* ------------------------------------------------------------------
* CP/M Version Notes: This operating system requires that you
* identify the system function needed by specifying a "function
* number". See the equivalent chapter 5 program for details.
* This program needs an additional EQUate for the SPACE assignment.
* ==================================================================
*                        S E T U P - B L O C K
* ------------------------------------------------------------------
CARRIAGE$RETURN          EQU     13
SPACE                    EQU     32
OPERATING$SYSTEM         EQU     5
                         ORG     100H
                         JP      STACK
                         ORG     150H
STACK:                   LD      SP,$-2
* ==================================================================
*                        M A I N - B L O C K
* ------------------------------------------------------------------
START:                   CALL    INPUT$ROUTINE
                         CP      CARRIAGE$RETURN
                         JP      Z,FINISH
                         CP      SPACE
                         JP      NC,PRINT$CHARACTER
                         LD      A,BELL
PRINT$CHARACTER:         CALL    OUTPUT$ROUTINE
                         JP      START
* ==================================================================
*                        E N D - B L O C K
* ------------------------------------------------------------------
FINISH:                  JP      0            ;Reboot operating system
* ==================================================================
```

```
*                      I N P U T - R O U T I N E
* -----------------------------------------------------------------
* Notes:  We have to use a "wait for input" loop here. With this
* system  (CP/M)  it is necessary to preserve the contents  of  the
* registers before using the operating system calls.
* -----------------------------------------------------------------
INPUT$ROUTINE:          PUSH BC ! PUSH DE ! PUSH HL  ;Preserve reg's
INPUT$ROUTINE$1:        LD      E,0FFH      ;Signifies console input
                        LD      C,6     ;Direct console I/O function
                        CALL    OPERATING$SYSTEM
                        CP      0                ;0 = no key pressed
                        JP      Z,INPUT$ROUTINE$1
                        POP HL ! POP DE ! POP BC  ;Restore registers
                        RET
* =================================================================
*                      O U T P U T - R O U T I N E
* -----------------------------------------------------------------
OUTPUT$ROUTINE:         PUSH AF ! PUSH BC ! PUSH DE ! PUSH HL
                        LD      E,A       ;transfer is in E register
                        LD      C,2         ;Console output function
                        CALL    OPERATING$SYSTEM
                        POP HL ! POP DE ! POP BC ! POP AF
                        RET
* =================================================================
```

7
ADDRESSING

Addressing refers to the way in which we specify the *location* of the operand (the byte(s) on which an instruction will operate). In this chapter we look briefly at some of the addressing *modes* you need to be familiar with. In each case we will consider the addressing mode in a general sense, to give you a feel for the overall ideas, then we will show you how the specific addressing modes available on the Z-80 fit into this framework.

IMPLIED ADDRESSING

Most processors have instructions that enable specified internal registers to be incremented or decremented. As an example the Z-80 uses INC B to increase the value of the B register by one. The instruction when assembled results in a single object code byte. The 'address' of the operand (which in this case is the B register) is specified within the op code. This form of addressing is termed '*implied*' or '*implicit*'. It is used in instructions such as register-to-register transfers, register increments and register decrements. Other specific examples are LD r, r1; ADC A, s; SUB s; XOR s; OR s.

Note:
Zilog do in fact make a distinction between instructions, such as the set of arithmetic operations where the accumulator is always implied as the destination register, and those cases where the op codes contain 'bit codes' to specify the registers. The latter form of addressing is given the name 'Register addressing'. This distinction is not always made by other microprocessor manufacturers and for normal use it is satisfactory to regard both forms as examples of implied addressing.

AN EXAMPLE

It is useful to examine one example of register addressing in detail to see just how the Z-80 implements it. The instruction LD r, r1 will copy the contents of register r1 into register r, where r and r1 may be the accumulator (register A), or B, C, D, E, H, or L registers.

When such an instruction is assembled it results in a single byte of object code, with bit 7 being 0 and bit 6 set to 1. The remaining six

bits of the object code byte depend on which registers are involved in the transfer as diagram 7.1 illustrates.

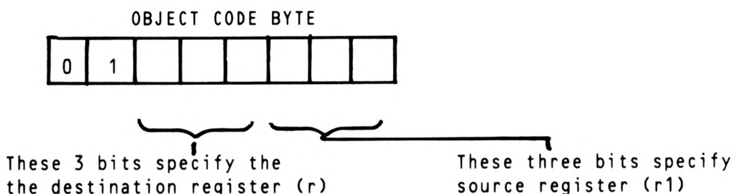


Figure 7.1: **Object code byte for a LD r, rl instruction**

The source and destination registers are each coded using three bits according to the following scheme:

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

The op code of the instruction thus varies depending on which particular registers are specified. The instruction LD A, C will load the accumulator with the contents of the C register. The three-bit code for the source register is 001, and the equivalent destination register code is 111. The instruction will thus be assembled to the single byte whose value is 01111001 binary, which is 79_{hex}. Try working out the op codes using other registers, then check your answers by using the instruction set listing.

IMMEDIATE ADDRESSING

If an instruction uses immediate addressing then it gets its operand byte(s) from the location or locations immediately following the op code in memory. One example is in the loading of constant values into a register.

These instructions, when assembled, result in two bytes of object code being produced; the op code itself followed by the data value. As we have seen previously the Z-80 has instructions which load register pairs with 16-bits of data, resulting in three bytes of object code being produced when the instructions are assembled – the op

code byte plus the two data bytes. Such instructions are often, somewhat pedantically, referred to as using *'extended immediate addressing'*.

A point of confusion sometimes arises with the JP instruction. This is normally thought of as an instruction that specifies a jump to a memory location specified by a two-byte address. Such jumps are often thought of as 'absolute addressing jumps'. In the Z-80 literature such jumps are classed as immediate addressing instructions. The reason is that although they do specify an 'absolute jump' to a given memory location, the actual data transfer takes place by transferring the bytes immediately following the op code byte to the program counter. If you compare this mechanism with LD B, data, you will appreciate that in terms of data movement the 'immediate addressing' label is perfectly valid.

ABSOLUTE ADDRESSING

As we have mentioned, absolute addressing indicates the specification of a memory byte using a full 16-bit address. On the Z-80 such instructions consist of the op code, which may be one or two bytes long, followed by the two byte address giving the location of the operand (POKE address, value could be a classed as a typical *absolute addressing* BASIC statement). This form of addressing is frequently called *extended addressing* in Z-80 literature.

ZERO PAGE ADDRESSING

If we provide an absolute reference to an address within the first 256 bytes of memory, then we only need one byte for the address (ie. addresses from 00_{hex} to FF_{hex}).

This form of addressing is used to good effect on some processors, but on the Z-80 only a limited version is available involving the specialized RST instruction. Zero page addressing is sometimes called *short addressing*.

RELATIVE ADDRESSING

Instead of an address we give a displacement to be added to the value already in the program counter. Such displacements are restricted on 8-bit micros because they have to be specified with one byte. The form of the displacement on the Z-80 is called 'two's complement', and this method of representing a number is dealt with in chapter 8.

When you use relative addressing in your programs you will not normally have to calculate the displacement, your assembler will do it for you. If by chance you try to jump to a location whose displacement cannot be specified with one byte then the assembler

will make sure you realize by giving an appropriate error message. Programmers who have written assembly language programs using hexadecimal op-code entry have frequently been known to say extremely unkind words about relative jump instructions.

COMPUTED ADDRESSING

Up to now the addressing modes we have looked at may be regarded as 'static' – once the program has been completed, the memory locations upon which instructions will operate are fixed, completely defined by the instructions you have selected. Computed addressing enables the address of an operand to be *computed* at run time and falls into two categories:

- Indexed addressing
- Indirect addressing

INDEXED ADDRESSING

Indexed addressing uses an address that is obtained by modifying a specified *base address* given in the program. The base address is modified by providing a *displacement* which is added to the base value to provide the final address used by the instruction.

As an example let us suppose you have a table of twenty single byte data items held in memory, the lowest byte is labelled 'BASE' and this base address is loaded into index register IX. If we write the displacement as 'n' then the instruction LD A, (IX+n) will access the base value if n is 0, the byte above this if the displacement is 1, and so on. In general it will access the *n*'th item of the table. Figure 7.2 illustrates the general idea.

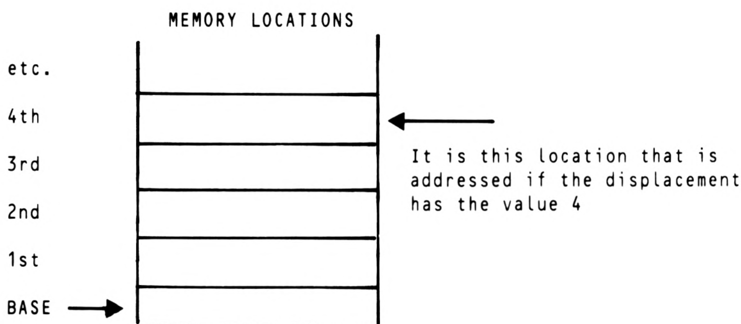


Figure 7.2: **Schematic description of Indexed addressing**

You've probably used similar ideas in your BASIC programs:

```
FOR I% = 1 TO 9 : PRINT X(I%) : NEXT I%
```

When `I%=4` you are referencing `X(4)` etc. Indexed addressing is particularly useful for accessing succesive data elements from tables or blocks of data.

You may be surprised that the displacement shown in the Z-80 instruction above is held within the instruction, rather than being specified by say, the contents of a register. On the face of it, the displacement is fixed when the instruction is assembled, which restricts the usefulness of the indexing capabilities. This is a fair criticism, the indexing facilities are not as generally useful as one would like, but, nevertheless they can still be used to advantage. We will see later that it is possible to alter the value of the displacement by modifying the contents of the displacement byte during program execution. Such 'tricks' are not generally used although they do appear on occasion and should help to give you an idea of what can be done.

INDIRECT ADDRESSING

The term *indirect addressing* refers to the concept of using one address to 'point' to another. Imagine a data file of one thousand items, each with a record length of 128 bytes. How can we sort these items using the 'field' – or part of the record – lying between bytes 6 and 20 of *each* record? An easy approach is to load just the fifteen bytes of interest from each record into a '*vector*', say `INDEX$()` and in addition create a '*tag vector*', `I%()`, that holds each record's 'record number'. Before sorting `I%()` will simply contain the numbers 1 to 1000 in order. We then perform a sort and rearrange the `I%()` vector to 'mirror' any physical (or logical) changes made in the index vector. After sorting the vector `INDEX$()` will be in the required order but `INDEX$(5)` will not now necessarily relate to the fifth record of our data file. By searching through `INDEX$()` we effectively move through the data file in the required sorted order but this is of little use unless we can access the corresponding data record. How do we access the records? We use the '*tag vector*' `I%()` that holds the corresponding original record numbers. Thus the record number of the first record in the sorted order, whose index value is `INDEX$(1)` is found from `I%(1)`. Similarly the `X`th item in the sorted order is obtained from `I%(X)`. We use the tag vector `I%()` to 'point' to the records in the data file. When we use the BASIC statement `GET #1, I%(5)` to get the fifth record in the new sorted order, we are specifying its address indirectly. We are saying in effect that the 'address' of the record in question is held in the variable `I%(5)`.

Indirect addressing in an assembly language instruction involves the same general idea as our BASIC example. We do not specify the operand's address, we specify the locations from which the address may be obtained. In the case of the Z-80 processor a form of indirect addressing often called '*register indirect*' is available. It is a *register pair*, rather than a pair of memory locations, that holds the required address.

BIT ADDRESSING

Bit addressing relates to the technique the Z-80 uses to access specified bits in a register or memory location. It is not generally thought of as a 'proper' addressing mode, but since it is mentioned in the literature we have included it for completeness. The Z-80 has special instructions for setting, resetting and testing specific bits within an internal register or a memory byte. The byte itself may be addressed by one of three 'conventional' addressing modes, namely indexed, indirect (ie. register indirect) or register addressing.

COMBINED ADDRESSING MODES

Some instructions involve more than one operand. In these cases the addressing mode used to get the source data is not necessarily the same as the addressing mode used to transfer the necessary information to its destination. A typical example is the loading of an indexed addressed memory location with a byte of immediate data: the instruction LD(IY+d), 8 will load the value 8 into the memory location whose address is given by the contents of register IY plus the displacement value 'd'. The instruction is using immediate addressing to obtain the source data, ie. the number 8, and it is using indexed addressing to send this value to its destination.

- Occasionally you will come across variants of the basic addressing modes that have been discussed. This is particularly true if you work with other processors. It is usually possible to understand the meaning of such terms by concentrating on the instruction itself; examine what the instruction does, ask yourself where it gets its data from, and identify the destination of the results of the instruction etc. You will usually be able to relate the general addressing mode to one of those that have been dealt with in this chapter.

THE 'CONNECT FOUR' GAME

We are going to illustrate some of the various addressing schemes by examining one way to represent the game 'Connect Four'. This also gives us the chance to examine some of the logical instructions available on the Z-80.

In this game two players have sets of coloured counters which are dropped (one at a time by alternate players) into one of seven columns. The first player to get four counters in a vertical, horizontal, or diagonal line has won the game. The game is played on an upright hollow board of six rows and seven columns. We are going to look at how such a game could be represented within a computer and we will start out with a list of the main requirements:

- A subroutine to set up (clear) the board representations
- A subroutine to get players move (ie. a column number)
- A subroutine to check that move is valid
- A subroutine to 'make the move' on the computer's boards
- A subroutine to identify change of player for next move

We need to define how we are to represent the game internally. We shall represent each player on a separate board created by seven bytes of memory, each of which will constitute one column of the games 'board'. As you look at the memory description in figure 7.7 bear in mind the the boards are twisted sideways in memory. The base locations that we have labelled are the 'column 0' bytes. As the game is played column 0 would be on the left hand side, column 6 on the right (see figure 7.8). We've numbered the seven columns from 0 to 6 because of the way we shall use indexing to access them. The six rows however, have been numbered from 1 to 6 because the row number then represents the 'bit position' within the byte.

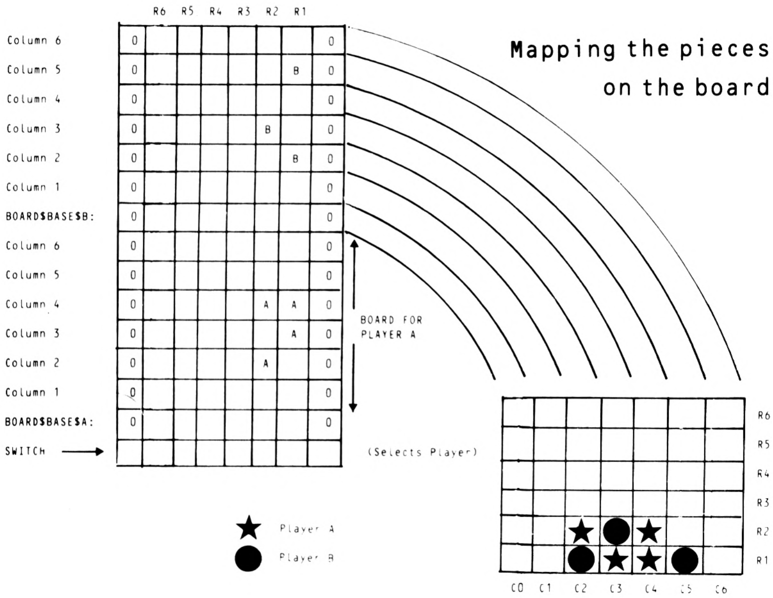
The presence of a counter in a certain position will be indicated by setting the equivalent 'bit' to 1. Our bytes are eight bits wide and (for reasons that we shall explain later) we will use the inner six bits of the bytes. We will also select one byte of memory to act as a player switch and shall change its value with each move to identify which player is making a move. Seven bytes will be used to count how many 'pieces' have been placed in a given column and a further seven bytes used to identify the position of the last piece placed in a given column.

MEMORY CLEARING ROUTINE

We will, at the end of a finished program, use an assembler pseudo operation to reserve certain memory locations for use by our program. The operation is usually called 'reserve data storage space'. Our assemblers use the instruction DS N to reserve N memory locations. In our case this space will sit immediately above the actual program code, figure 7.7 shows how we have chosen to allocate its use.

ADDRESSING	MEMORY LOCATIONS								
Column 6	0							0	
Column 5	0							0	
Column 4	0							0	
Column 3	0							0	BOARD FOR PLAYER B
Column 2	0							0	
Column 1	0							0	
BOARD$BASE$B:	0							0	
Column 6	0							0	
Column 5	0							0	
Column 4	0							0	
Column 3	0							0	BOARD FOR PLAYER A
Column 2	0							0	
Column 1	0							0	
BOARD$BASE$A:	0							0	
SWITCH: →									(Selects Player)
Column 6	0							0	
Column 5	0							0	
Column 4	0							0	
Column 3	0							0	Number of counters in binary number form
Column 2	0							0	
Column 1	0							0	
COUNTERSINBASE:	0							0	
Column 6	0							1	
Column 5	0							1	
Column 4	0							1	
Column 3	0							1	Bits in position 0 are shifted to left to make a move
Column 2	0							1	
Column 1	0							1	
ROW$POINTER$BASE:	0							1	

Internal representation of the game board



We must write a subroutine to clear the area of memory assigned for the boards, and make the initializations needed to the switch byte (we will arbitrarily set this to 0 to indicate player 'A' and to FF_{hex} to indicate player 'B'). Seven bytes are initialized, starting at the location labelled ROW$POINTER$BASE, so that they contain the value 00000001 binary. We will be using an operation called a *left shift* to push those single bits from right to left as the game progresses.

On the Z-80 the index registers IX and IY are used to hold base addresses and *not* offset values. We already know that the indexed instructions offer the inclusion of a *displacement value* within the instruction itself. The instruction LD (IX + number), value will load the specified value into the memory location whose address is 'IX + number'. When assembled in memory, the layout of the instruction is:

```
  BYTE 1          BYTE 2          BYTE 3          BYTE 4
1st OP CODE     2nd OP CODE     DISPLACEMENT     LITERAL

 DD hex           36 hex          number          value
(    op code values      )
```

Notice that we have an instruction here that has a two-byte op code resulting in a total instruction length of 4 bytes. Let us use this instruction to create a simple loop that stores a constant value in a set of adjacent locations.

```
* ==================================================================
*                  Z 8 0  -  V E R S I O N  -  1
* ------------------------------------------------------------------
                LD   IX, BASE          ;Set up index register IX
                LD   C, n              ;Number of bytes
START:          LD   (IX+0), value     ;Value stored at address in IX
                INC  IX                ;Increase register IX  by 1
                DEC  C                 ;Decrease counter C
                JR   NZ, START         ;Back for next byte if C<>0
                                       ;(relative jump !)
* ==================================================================
```

Notice that within this loop we are essentially using the index register as a 'pointer' to the location in which we wish to store the data item. We are not therefore using 'indexing' in the true sense of our original definition but are in fact effectively using the IX register to specify an address which is then used to store the data.

If we wished to implement the variable displacement that we discussed in our general definition of indexing we could use the HL register pair to 'point' to the byte holding the displacement and modify it during execution by using a DEC (HL) instruction like this:

```
* ==================================================================
*             Z 8 0  -  V E R S I O N - 2
* ------------------------------------------------------------------
                LD   IX, BASE-1        ;Byte below base address
                LD   HL, TARGET+2      ;HL points to displacement
                LD   (HL), N           ;N is the number of bytes
TARGET:         LD   (IX+0), value     ;Run time modified displacement
                DEC  (HL)              ;Decrease displacement
                JR   NZ, TARGET        ;Back for next byte if disp <>0
* ==================================================================
```

The following coding uses two loops, one to initialize with zeros the memory between the byte labelled `COUNTERS$IN$BASE` and the top of board 'B', the other to initialize the seven row pointer bytes. At the end of the routine we also set B and D registers to zero. The reason for this will become apparent later.

```
* ==================================================================
*          C L E A R - M E M O R Y - S U B R O U T I N E
* ------------------------------------------------------------------
CLEAR$MEMORY:   LD      IX, COUNTERS$IN$BASE
                LD      C, 22
C$M$1:          LD      (IX+0), 0         ;Set these bytes to 0
                INC     IX
                DEC     C
                JR      NZ, C$M$1
                LD      IX, ROW$POINTER$BASE
                LD      C, 7
```

```
C$M$2:          LD      (IX+0), 1           ;Set these bytes to 1
                INC     IX
                DEC     C
                JR      NZ, C$M$2
                LD      B, 0             ;We set B and D to 0 in order
                LD      D, 0             ;to use ADD HL, BC etc. later
                RET
* ==============================================================
```

GET MOVE SUBROUTINE

We use a system input routine to collect a column number in the accumulator. One immediate problem is that the ASCII character codes for the numbers 0 to 9 on the keyboard are not the numeric values of the numbers themselves. The values are as follows:

DECIMAL	BINARY	ASCII VALUE
0	0000 0000	0011 0000
1	0000 0001	0011 0001
2	0000 0010	0011 0010
3	0000 0011	0011 0011
4	0000 0100	0011 0100
5	0000 0101	0011 0101
6	0000 0110	0011 0110
7	0000 0111	0011 0111
8	0000 1000	0011 1000
9	0000 1001	0011 1001

– the 8-bit numbers have been slightly separated into two 4-bit 'nibbles' here for clarity. To convert the ASCII form to a real binary equivalent of the input number we need to set the upper four bits of the ASCII form to zero. This can be accomplished by using an 'AND' operation. Essentially two bytes, one of which is the accumulator, are compared bit-by-bit. If both bits are set to 1 then the corresponding accumulator bit is set to 1, otherwise the accumulator bit is set to 0. Figure 7.3 shows the effect on the ASCII code for the number 9:

```
ACCUMULATOR     0 0 1 1 1 0 0 1     < ------ ASCII '9'
OTHER BYTE      0 0 0 0 1 1 1 1     < ------ 'MASK'
                ---------------
RESULT          0 0 0 0 1 0 0 1     < ------ REAL '9'
```

Figure 7.3: **The effect of an 'AND' operation on an ASCII character**

The value we compare against is often called a 'mask' – for obvious reasons. On the Z-80 several addressing modes are available with the AND operation. We shall use an immediate addressing mode to compare the accumulator with $0F_{hex}$ (00001111 binary). The

mnemonic will thus take the form AND OFH. Having obtained a proper numeric representation of the input character we store it in the C register by using a LD C, A instruction. We then have the column number for the user selected column in the C register. To collect a character from the keyboard, mask it, and store it in the C register we use the following code:

```
CALL  INPUT$ROUTINE
AND   OFH
LD    C, A
```

COMPUTING THE OFFSET INTO THE BOARD

The offset into the boards is dependent on whether player 'A' or player 'B' is being dealt with. We use the value held in the switch byte to decide which board requires updating.

As one of several alternatives we load the accumulator with the contents of the switch byte and then add the contents to itself. This sets or clears the sign flag which is then used to add, or not add, the offset for board 'B'. We have chosen to store the result in the E register.

```
* ================================================================
*          G E T - M O V E - S U B R O U T I N E
* ----------------------------------------------------------------
GET$MOVE:       CALL    INPUT$ROUTINE
                AND     OFH             ;Mask upper four bits
                LD      C, A            ;Save column no. in C register
                LD      E, A            ;and as the board 'A' offset
                LD      A, (SWITCH)
                ADD     A
                JP      M, G$M$1
                LD      A, E            ;Get column number back
                ADD     7               ;Board 'B' additional offset
                LD      E, A            ;Replace offset value in E
G$M$1:          RET
* ================================================================
```

MOVE VALIDITY CHECK

On most microprocessors it is possible to shift bytes and registers to the left or right. The Z-80 has instructions to perform various shifts and we will make use of the instruction SLA which is an *arithmetic shift left*. Our row pointer bytes are initialized to the value

00000001 binary by the 'clear memory' coding. Figure 7.4 shows the effect of such a shift on the accumulator.

```
0 0 0 0 0 0 0 1   <-- initial value of accumulator
0 0 0 0 0 0 1 0   <-- accumulator after  one  SLA instruction
0 0 0 0 0 1 0 0   <-- accumulator after two SLA instructions
```

Figure 7.4: **Effect of left shift on the accumulator contents**

The bit at the right hand side is always set to zero. The bit on the left hand side is shifted into the carry. If we used the instruction SLA A then we would perform the above shift on the contents of the accumulator.

We want to load the accumulator with any one of seven bytes, depending on the value of the C register. On the Z-80 we can do that in the following manner. We load the HL register pair with the address whose label is ROW$POINTER$BYTE then add to it the contents of register C. A specific instruction for adding pairs of registers proves useful: we use ADD HL, BC to add the contents of BC to the address in HL. Since we have previously set B to zero we are effectively just adding C to the HL contents. Having done this we use SLA, A to shift the contents of the accumulator to the left. Think about this carefully, use figures 7.7 and 7.8 if you find it difficult to picture.

After this instruction has been performed, the single bit will be in the bit position corresponding to the board position to be updated by this move. This representation has been arranged so that if the bit has been shifted to the bit 7 position then the move is illegal because the column already has six pieces in it. We can tell this because the SLA instruction on the Z-80 affects the carry, the parity, the zero and the sign flags. The sign flag is used to determine the status of bit 7. The type of coding we use is shown below.

```
* ==================================================================
*          C H E C K - M O V E - S U B R O U T I N E
* ------------------------------------------------------------------
CHECK$MOVE:     LD      HL, ROW$POINTER$BASE
                ADD     HL, BC          ;Effective HL+C since B=0
                LD      A, (HL)         ;Image of column's last move
                SLA     A               ;Left shift
                RET
* ==================================================================
```

MAKING THE MOVE

After the 'check move' subroutine has been performed we will have an image of the new move held in the accumulator. The first step therefore is to store the contents of the accumulator back in the location used in the 'Check Move' subroutine. Following this it is necessary to add the new move into the appropriate board column. Figure 7.5 illustrates the effect we wish to obtain to 'create the new move'.

```
BYTE - ROW$POINTER$BASE+C - 0 0 0 0 0 1 0 0 <-- image of the new move
                                                 in the accumulator
BYTE - BOARD$BASE$A+B  -    0 0 0 0 0 0 1 0 <-- current column state
                            ---------------
RESULT
NEEDED IN ACCUMULATOR  -    0 0 0 0 0 1 1 0 <-- required new state
```

Figure 7.5: **creating a new move**

Another logical function exists, called OR that tests the accumulator against another specified byte. It will set any accumulator bit to 1 if either or both of the respective bits in the accumulator *or* if the other byte specified is set to 1.

The Z-80 has an 'OR' instruction which OR's the accumulator with another specified byte. We are going to use the instruction to OR the image of the current state of the column in question with the new move present in the accumulator. The updated column will then be replaced in its correct memory position. Having done this we increase the value of the corresponding numerical count of the number of pieces in the column. The code to store the new row position byte, create the new move in memory and update the numeric count is shown below:

```
* ====================================================================
*           M A K E - M O V E - S U B R O U T I N E
* --------------------------------------------------------------------
MAKE$MOVE:      LD      (HL), A         ;Replace updated column image
                LD      HL, BOARD$BASE$A
                ADD     HL, DE            ;Now HL points into boards
                OR      (HL)              ;Create new board image
                LD      (HL), A           ;and replace in memory
                LD      HL, COUNTERS$IN$BASE
                ADD     HL, BC           ;HL now points to count byte
                INC     (HL)             ;Increase numeric count
                RET
* ====================================================================
```

CHANGING THE PLAYER

We change players by changing the value of the byte we have labelled SWITCH. We set it to zero when we perform the clearing of memory. After each move we want to change the value, so that it alternates. We have seen examples of AND and OR as logical functions, another logical function is called 'exclusive OR'. This is similar to the OR described earlier, except that if both bits being tested are high (ie. 1) then the accumulator bit will be set to 0 and not 1. Figure 7.6 shows this:

```
BYTE BEING TESTED        0 0 1 1 0 0 0 0
ACCUMULATOR              1 0 1 0 0 0 1 0
                         ---------------
RESULT                   1 0 0 1 0 0 1 0
```

Figure 7.6: **Example of the effect of an 'exclusive OR'**

We could use such an instruction, called XOR, to change the switch from 00000000 binary to 11111111 binary. If the switch already contains 11111111 binary then the XOR 0FFH instruction will change it back to 00000000 binary. The Z-80 does however have an explicict instruction CPL to complement the accumulator (ie. set all ones to zeros and all zeros to ones) and the following example uses this:

```
* ===================================================================
*          C H A N G E - P L A Y E R - S U B R O U T I N E
* -------------------------------------------------------------------
CHANGE$PLAYER:  LD      A, (SWITCH)     ;Get current player
                CPL                     ;Complement the 'switch' byte
                LD      (SWITCH), A     ;Changed for next player
                RET
* ===================================================================
```

SOME NOTES ON THE DESIGN

The internal representations of the boards themselves can be examined in several ways. We could write a routine to display the contents of the bytes in binary form, we could use the system monitor to examine the bytes in question, or we could use a dynamic debugging tool (such as CP/M's DDT program) to examine memory areas during execution of a program.

Most of the complexity of our example lies in the representation of the game, not in the individual instructions used. We would like to explain a few points about our choice of representation:

- We have chosen to represent each player individually because when it comes to devising a strategy with the computer playing it

will be useful to have such separation. Routines that for one player find the 'best move to make' can then be used on the other board to find the 'most appropriate blocking move'.

The visual display of the game would of course represent both players on the same board, it is only the internal representation that uses separate boards.

- The inner six bits were chosen specifically so that the shift instruction will, once a column is 'full up', push that single bit representation into bit position 7. We make use of the fact that this results in a flag being set and are thus able to detect and reject illegal moves, ie. moves into a full column.
- The co-ordinates of the column and row of a given move could be passed to a graphics routine that would handle the visual display of the 'real', or displayed, board.

Some of the ideas presented will take time to digest. The essential idea should be reasonably easy to follow, and routines such as the 'clear memory' subroutines should also not take long to understand. The instructions that make a move and update an internal board depend on an understanding of how we are representing the game – this may take time to absorb. Don't worry unduly if the representation of the game seemed a bit complicated, concentrate on easier sections. The main thing is to understand what we mean by 'addressing' and appreciate the main differences between the various modes.

LINKING AND TESTING

We have now developed some routines applicable to the game 'Connect Four'. Obviously these are just first steps in such a development, but, even at this stage the routines must be checked to ensure that they work. A common technique (and one which we use frequently) is to write short 'test-bed' controller routines, ie. a short patch of code is written that uses the subroutines under development so that we can check their performance. To illustrate how we go about this we have written a short routine to test the subroutines we have been looking at.

Our first job was to sketch out a brief 'controller structure' using a Warnier diagram as shown in figure 7.9.

We should point out that the general ideas of the game were in fact also examined in Warnier form first, that's how we produced our initial objectives. Most of the statements in figure 7.9 correspond to subroutines that we have already written but we would like to make the following additional observations. The 'end of game' statements

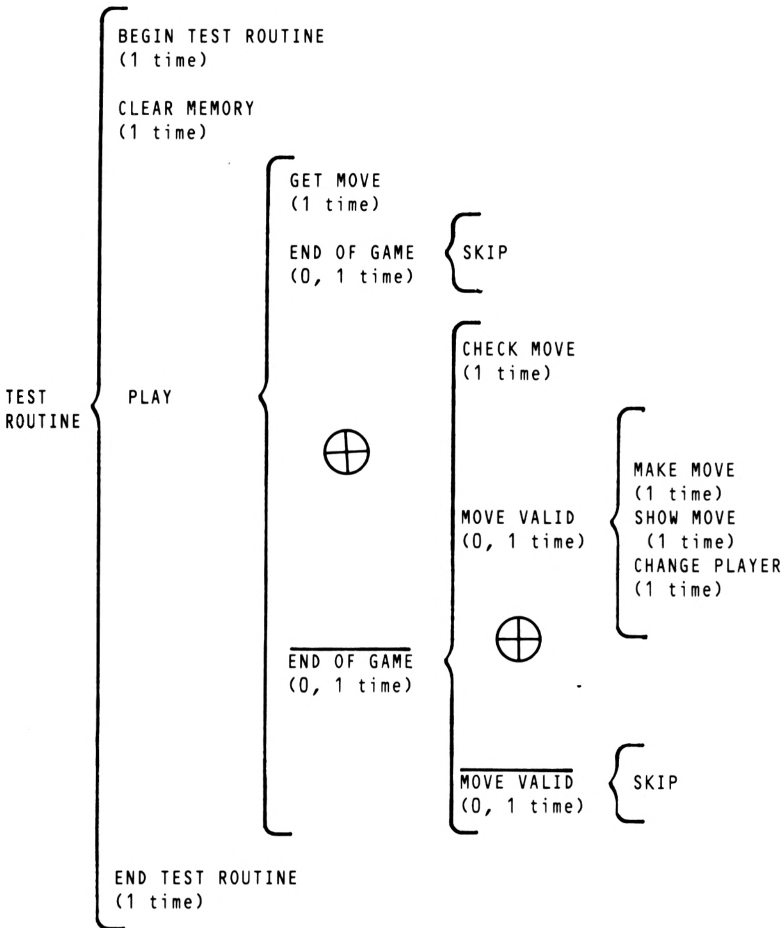


***Figure 7.9:* Warnier representation of the test bed control routine**

imply that we can detect the end of the game – this we cannot do since there is, as yet, no playing strategy available. With this in mind we must be satisfied with either testing the routines using an 'infinite loop', or with terminating the controller program when a particular keyboard character is detected. We choose the latter option and will arbitrarily use a carriage return to signify the end of game condition.

We also need some temporary 'show move' code, for illustration purposes we adopt a simple solution: we just output the row number that represents the position in the given column that the latest move will occupy. In writing the controller routine the aim is only to test the subroutines we have written.

To summarise: the controller block starts by clearing the memory, then we collect a character with the 'get move' subroutine. If a carriage return is detected we end the program, otherwise we check the move. If the move is illegal (ie. a move to a column that is full) we ignore it, otherwise we make the move on the internal boards and display it by outputting the 'row number'. Lastly we change the player before returning to collect another move. We have not included a check to ensure that any column number entered lies between 0 and 6 since this method of identifying a move is really only applicable during the development stage where such checks are not absolutely necessary.

We have kept the 'test bed' program listing separate from the listings of the subroutines developed. This should make it easier for you to see the basic ideas behind the controller routine, and also allows you to view the subroutines that we have developed 'in isolation'. If problems occur, one useful tip is to modify the controller routine to eliminate calls to any suspect subroutines. To be safe you may prefer to start with a controller routine that just calls the 'clear memory' subroutine. Once this is working satisfactorily the 'get move' subroutine can be included. In this way you can build up the controller routine one piece at a time.

The layout of the 'test bed' program is as follows: We start with a 'set up' block, defining equates, initializing stacks, etc., as required. Next comes the controller routine which makes calls to the various subroutines that have been developed. Immediately following this we place the subroutines that we wish to test, including any other necessary routines such as input / output routines. Lastly we identify our data storage areas which sit on top of the program.

```
* ====================================================================
*C O N N E C T - F O U R - T E S T - B E D - P R O G R A M
* ====================================================================
*                   SET  UP  BLOCK
* --------------------------------------------------------------------
CARRIAGE$RETURN     EQU  13
OPERATING$SYSTEM    EQU  5                  ;Entry point
                    ORG  100H
                    JP   STACK
                    ORG  150H
STACK:              LD   SP, $-2
* ====================================================================
*                   CONTROLLER - ROUTINE
* --------------------------------------------------------------------
                    CALL CLEAR$MEMORY
PLAY:               CALL GET$MOVE
                    LD   A, C
                    CP   CARRIAGE$RETURN
                    JP   Z, FINISH          ;End of game
                    CALL CHECK$MOVE
                    JP   M, PLAY           ;Illegal move so ignore it
```

```
                CALL MAKE$MOVE
                LD   A, (HL)       ;Get row number for display
                OR   00110000B    ;Convert to  ASCII equivalent
                CALL OUTPUT$ROUTINE        ; ie. we 'Show move'
                CALL CHANGE$PLAYER
                JP   PLAY                  ;Back for next move
* ==================================================================
FINISH:         JP   0                 ;Re-boot operating system
* ==================================================================
*            IN  THIS  AREA  PLACE   SUBROUTINES
*            SHOWN  IN  THE   SEPARATE   LISTING
*            (INCLUDE ANY I/O ROUTINES REQUIRED)
* ==================================================================
*           W O R K S P A C E - D E F I N I T I O N S
* ------------------------------------------------------------------
ROW$POINTER$BASE:   DS   7             ;Bit marked 'counter height'
COUNTERS$IN$BASE:   DS   7             ;Numeric form 'counter height'
SWITCH:             DS   1             ;Identifies current player
BOARD$BASE$A:       DS   7             ;Player A's board bit map
BOARD$BASE$B:       DS   7             ;Player B's board bit map
* ==================================================================
```

```
* ==================================================================
*          C L E A R - M E M O R Y - Z 8 0 - V E R S I O N
* ------------------------------------------------------------------
CLEAR$MEMORY:   LD      IX, COUNTERS$IN$BASE
                LD      C, 22
C$M$1:          LD      (IX+0), 0           ;Set these bytes to 0
                INC     IX
                DEC     C
                JR      NZ, C$M$1
                LD      IX, ROW$POINTER$BASE
                LD      C, 7
C$M$2:          LD      (IX+0), 1            ;Set these bytes to 1
                INC     IX
                DEC     C
                JR      NZ, C$M$2
                LD      B, 0           ;We set B and D to 0 in order
                LD      D, 0           ;to use ADD HL, BC etc. later
                RET
* ==================================================================
*          G E T - M O V E - Z 8 0 -  V E R S I O N
* ------------------------------------------------------------------
GET$MOVE:       CALL    INPUT$ROUTINE
                AND     0FH           ;Mask upper four bits
                LD      C, A          ;Save column no. in C register
                LD      E, A          ;and as the board 'A' offset
                LD      A, (SWITCH)
                ADD     A
                JP      M, G$M$1
                LD      A, E          ;Get column number back
                ADD     7             ;Board 'B' additional offset
                LD      E, A          ;Replace offset value in E
G$M$1:          RET
```

```
* ====================================================================
*          C H E C K - M O V E - Z 8 0 - V E R S I O N
* --------------------------------------------------------------------
CHECK$MOVE:     LD      HL, ROW$POINTER$BASE
                ADD     HL, BC          ;Effective HL+C since B=0
                LD      A, (HL)         ;Image of column's last move
                SLA     A               ;Left shift
                RET
* ====================================================================
*          M A K E - M O V E - Z 8 0 - V E R S I O N
* --------------------------------------------------------------------
MAKE$MOVE:      LD      (HL), A        ;  Replace updated column image
                LD      HL, BOARD$BASE$A
                ADD     HL, DE              ;Now HL points into boards
                OR      (HL)                ;Create new board image
                LD      (HL), A             ;and replace in memory
                LD      HL, COUNTERS$IN$BASE
                ADD     HL, BC             ;HL now points to count byte
                INC     (HL)               ;Increase numeric count
                RET
* ====================================================================
*          C H A N G E - P L A Y E R - Z 8 0 - V E R S I O N
* --------------------------------------------------------------------
CHANGE$PLAYER:  LD      A, (SWITCH)     ;Get current player
                CPL                     ;Complement the 'switch' byte
                LD      (SWITCH), A     ;Changed for next player
                RET
* ====================================================================
```

8
REPRESENTING NUMBERS

We very much take for granted the facilities offered by high level languages for adding, subtracting, multiplying and dividing and so on, and an appreciation of how languages, such as BASIC, actually perform the 'arithmetic' is useful in gaining insight into the problems involved when providing such facilities. Our first job is to look, in a general sense, at the way we represent numbers inside a computer and it is to this problem that we first turn our attention.

The 8-bit processors, including the Z-80, only have instructions for performing elementary addition and subtraction. To provide anything more sophisticated requires us to program the more complex procedures in terms of the simple operations that the processor can perform. We look first at some general ideas and then move on to relate these ideas to some assembly language routines.

REPRESENTING INTEGERS

With the eight bits of a single byte we can represent numbers from 00000000 binary to 11111111 binary (0 to 255 decimal). To represent larger numbers we must use more bits. By using *two* bytes for the representation we can deal with integer numbers up to the value 65536 , i.e 1111 1111 1111 1111 Binary. The magnitude of a number that can be represented in this way is therefore limited by the number of bytes we choose to assign to its representation.

This form of representation is called 'unsigned binary'. If we wish to allow for the occurrence of negative numbers it is necessary to make provision within the representation of the number to indicate whether it is positive or negative. This can be done by using one bit as a 'sign' bit. By convention we use the most significant bit, i.e the left-most bit:

- We set the bit to 0 to represent a positive number and to 1 to indicate a negative number .

An 8-bit 'signed binary' number will therefore have only seven bits for the numerical value. As an example, decimal 5 (101 Binary) can be represented as follows:

```
+5  Signed binary form    = 0 0000101
-5  Signed binary form    = 1 0000101
                            ↑
      Leading bit used to represent the sign of the number
         (This has been separated only for clarity)
```

By using a suitable number of bytes, and using one bit as a sign bit we can represent both positive and negative numbers of any magnitude. To a large extent our problems of representation must surely be over? If we just wanted to represent the numbers then this would in fact be partly true. The problem is that we don't just want to represent the numbers, we want to manipulate them, to add and subtract and so on. Let us add two positive numbers 4 and 5 as an example.

```
+4    is          00000100
+5    is          00000101
                  --------
Result            00001001  represents '9' (which is correct)
```

Now we try adding the two numbers -4 and +5:

```
-4    is          10000100
+5    is          00000101
                  --------
Result            10001001  represents '-9' (which is incorrect)
```

The result we should have obtained is +1 so clearly a problem exists with the representation or in the way we are using it. The solution lies in using what is called 'two's complement' representation. In this form, positive numbers are represented in the usual signed binary form, but for negative numbers we take the 'unsigned binary' form and complement it, i.e turn all the 1's into 0's and all the 0's into 1's (often called the 'one's complement' form). Having done this we add 1 to the result to obtain the final 'two's complement' representation. It can be shown that if we use this representation the results of arithmetic operations, including the sign, will come out correctly (*there is one proviso which will be covered shortly*).

Let's work through some examples to get the general idea. First let's redo the addition of -4 to +5 that we tried earlier. +5 being a positive number is represented in usual signed binary form but we must convert -4 to its two's complement in the manner mentioned above. When we have obtained the correct representation we will retry our example to see what result we get. The details are shown in figure 8.1.

```
Conversion to the two's complement form:

00000100 is binary 4
11111011 One's complement form of -4
11111100 Two's complement form of -4


Addition of the two's complement forms:

       11111100     -4    (two's complement form)
       00000101     +5    (two's complement form)
       ----------
(1)    00000001     Result +1 (which is correct)
 ↑     ----------
Carry flag is set in  this  example
```

***Figure 8.1:* Example of addition using two's complement arithmetic**

One of the rules of two's complement arithmetic is that the setting of the carry flag itself can safely be ignored.

If the magnitude of a result is too large to be expressed within the bits allotted for the representation of the numerical part of the number then it is possible for the sign bit to be changed accidentally. This is called 'overflow' and the effect is that an incorrect result is obtained.

The most obvious cause of such an error is an 'internal carry' from bit 6 to bit 7 as the following example shows:

```
                0 0111111      two's complement form of +63
                0 1000001      two's complement form of +65
                ---------
                1 0000000

                ↑
the 'sign' bit has been changed by
a carry from bit 6 to bit 7
```

Overflow can also occur when we add two negative numbers. In general it will occur when the result cannot be expressed in the seven bits available. It is obviously useful to be able to detect such a condition and most processors, including the Z-80, have an 'overflow' flag for this purpose.

Multiple-byte integers

The magnitude of the largest integer we can represent is governed by the number of bytes we use. We can show you the general idea by looking at how Microsoft's BASIC stores the 'integer variables'. When you write the BASIC statement `LET X% = 10`, the percent sign

indicates that an integer variable, X%, is being assigned the value 10. Can we write a BASIC program to look at the internal representation of such a number? Yes we can, and it is very easy to do. The function VARPTR(X%) is used to obtain the address of the variable X%. This byte and the contents of the following byte are examined using the PEEK() function, (after prior translation to hexadecimal form by use of the HEX$() function). For hex numbers less than 16, the HEX$() function returns only one character (eg: F rather than 0F), so we add the '0' to such numbers from within the program. The following program asks for an integer value and prints the hex form of the internal representation.

```
4 REM ==============================================================
5 REM          PRINT HEX REPRESENTATION OF INTEGER X%
6 REM ______________________________________________________________
10 INPUT'Please enter integer value'; X% 'Input an integer value
20 MSB$=HEX$(PEEK(VARPTR(X%)+1))'         Most significant byte
30 IF LEN(MSB$)=1 THEN MSB$='0'+MSB$
40 LSB$=HEX$(PEEK(VARPTR(X%)))'           Least significant byte
50 IF LEN(LSB$)=1 THEN LSB$='0'+LSB$
60 PRINT MSB$+LSB$'                       Shows how X% is stored
70 END
80 REM ==============================================================
```

Note:
The function VARPTR(), which is an abbreviation of 'variable pointer', is normally used to pass addresses of variables from a BASIC program to an assembly language routine.

If you run this program with the number 15 you will get 000F, which corresponds to the binary number 0000 0000 0000 1111. If you try -66 you will get FFBE and the following details show the reason why:

```
                        < MSB >    < LSB >

            66 =        0000 0000  0100 0010    Binary
            Complement  1111 1111  1011 1101
            add 1                          1

Two's complement form   1111 1111  1011 1110
Equivalent Hex form      F    F     B    E
```

FLOATING POINT NUMBERS

The representation of wide ranges of decimal numbers has its own special problems. The usual way of coping with wide variations in magnitude is to use scientific notation, for example 26063.15 can be represented as 2.606315×10^4, -0.000003415 can be written -3.415×10^{-6}. This gives a clue to providing a similar computer representation. We need to reserve bits for the mantissa, and further bits for the exponent. We also need to indicate the signs of each part of the

number. In scientific notation we 'normalize' the number by moving the decimal point to a position where the mantissa takes a value between 1 and 9.999. It turns out that for floating point representation it is better to move the 'binary point' to the far left of the number:

111.1101 is represented as .1111101 x 2^3

0.0000111 is represented as .111 x 2^{-4}

The general floating point format is based on the following type of scheme, *m* and *n* varying according to the number of bits chosen but the following schematic form will illustrate the essential idea.

1 bit Sign	*n* bits Exponent	1 bit Sign	*m* bits Mantissa

BINARY CODED DECIMAL NUMBERS

For some applications it is necessary to have complete numerical accuracy – an often quoted example is the use of computers in accountancy. For these applications an alternative representation called '*binary coded decimal*' or 'BCD' is sometimes used.

The principle is to code each digit separately, using as many bits as is necessary. Each digit reqires four bits with some combinations being unused:

BCD	Number
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010 – 1111	Unused codes

Two digits are packed into each byte, thus the amount of space a number will require is dependent on how many characters are present. The advantage of representing numbers in this way is that complete accuracy is obtained. The disadvantages are firstly that more memory is required to store the numbers, and secondly that arithmetic operations are slower.

Having briefly described some of the more common ways of representing numbers within a computer we turn our attention to some simple routines that use some of the forms we have discussed. Firstly, however, try this experiment – take a number and multiply it by 2, by 4, and by 8. Express the number and all of the products in their binary form and see what you notice about the bit patterns.

ARITHMETIC OPERATIONS

The basic arithmetic instructions available on the Z-80 processor are for addition and subtraction. Certain instructions allow 16-bit as well as 8-bit operands to be dealt with.

ADDITION

On the Z-80 the basic addition instructions take the general form 'ADD A, operand'. The function performed is to add the specified operand to the value already present in the accumulator and in symbolic form we can write A←A + operand. Various forms of addressing are possible when specifying the operand and some typical examples are shown below:

ADD A, 8
Adds the immediate value 8 to the accumulator, performing the function A←A + 8.

ADD A,B
Adds the contents of the B register to the accumulator thus performing the function A←A + B.

ADD A,(HL)
Adds to the accumulator the contents of the byte whose address is specified by HL like this A←A + (HL).

ADD A,(IX+d)
In the indexed addressing form the address of the byte to be added is found by adding a specified displacement to the address held in index register IX. The symbolic representation is A←A + (IX+d).

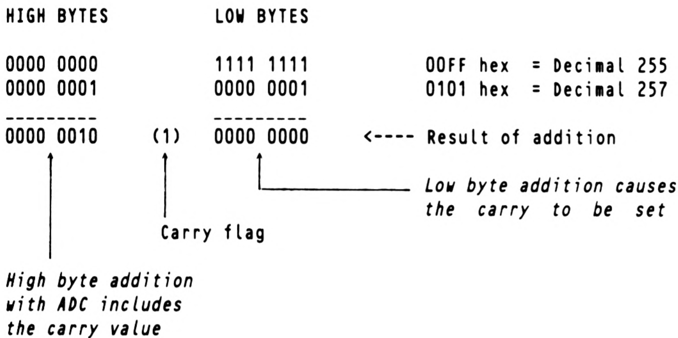
Instructions also exist that enable 16-bit operations and these use HL, IX or IY as destination registers. Typical examples are as follows:

ADD HL,DE
This adds the contents of the DE pair to the contents of HL thus performing HL←HL+DE.

ADD IX,BC
In a similar fashion to above this adds the contents of BC to the index register IX (IX←IX+BC).

All of the instructions looked at so far affect the flag register, but, because of the way the carry flag is used during multi-byte addition another addition instruction is made available. On the Z-80 the instruction '*add with carry*', ADC, will include in the 'addition' the carry flag value, i.e ADC A,B will perform the function A←A + B + carry.

The usefulness of this instruction can be seen from the following example. We will add two 'two byte numbers' 255 and 257 by adding the two low bytes first and then adding the two high bytes.



The addition of the low bytes causes a carry to occur. This carry must be taken into account when the high bytes are added and this is precisely what the Z-80 'add with carry' ADC instruction does. It is therefore performing the function that is shown by A←A + operand + carry. As a general rule, multi-byte addition is performed by using a normal addition instruction for the first – least significant – bytes, and using the add with carry instructions for succeeding bytes. Let us now write a short program to add the contents of two two-byte numbers held in locations labelled FIRST$NUMBER and SECOND$NUMBER. We assume, as is common convention, that the least significant bytes are held at the labelled address, with the most significant bytes following (as figure 8.2 shows). The result will be placed in two further locations the lowest of which is labelled RESULT.

The program takes the following form. The accumulator is loaded with the least significant byte of one of the numbers, (FIRST$NUMBER), and HL is loaded with the address of the least significant byte of the other number, SECOND$NUMBER. The instruction ADD A,(HL) is then used to add the byte specified by HL to the accumulator, thus performing the additions of the low bytes. This result is stored and the accumulator re-loaded with the *second* byte of the first number. HL is incremented so that it then points to

the second byte of the second number, and an ADC instruction used to obtain the high byte of the result which is then stored in address RESULT+1.

```
* ===================================================================
*                        Z-80 - 16  BIT - ADDITION
* -------------------------------------------------------------------
    LD   HL,SECOND$NUMBER       ;HL points to low byte of 2nd number
    LD   A,(FIRST$NUMBER)       ;Get low byte of 1st number in Acc.
    ADD  A,(HL)                 ;Add low bytes
    LD   (RESULT),A             ;Store low byte of result
    LD   A,(FIRST$NUMBER+1)     ;Get high byte of 1st number
    INC  HL                     ;Now points to high byte of 2nd number
    ADC  A,(HL)                 ;Add high bytes + carry
    LD   (RESULT+1),A           ;Store high byte of result
* ===================================================================
```

Because of the existence of double register addition instructions it is possible on the Z-80 to write a much simpler 16-bit addition program. DE and HL can be loaded directly with the numbers to add and an ADD HL,DE instruction used to perform the 16-bit addition with one addition instruction.

```
* ===================================================================
*                 Z-80 - ALTERNATIVE 16 BIT ADDITION
* -------------------------------------------------------------------
          LD   DE,(FIRST$NUMBER)         ;Load DE with 1st number
          LD   HL,(SECOND$NUMBER)        ;Load HL with 2nd number
          ADD  HL,DE                     ;Performs HL <--  HL + DE
          LD   (RESULT),HL               ;Store result
*
=====================================================================
```

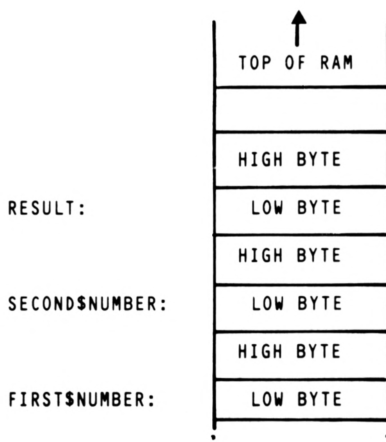


Figure 8.2: **Layout in memory of the two byte numbers**

SUBTRACTION

As with the addition instructions it is useful to have two types of subtraction, normal subtraction and subtraction with borrow. Normal subtraction (mnemonic is SUB) is used for the least significant or 'low bytes'; subtraction with borrow (mnemonic SBC) is used for the succeeding bytes. Most of the instructions in the following program are identical to the earlier addition program. If after the subtraction of the least significant bytes the carry flag has been set then this indicates that the value subtracted from the accumulator is greater than the accumulator value itself, i.e a borrow has occurred. The SBC instruction allows for this 'borrow' by including the carry flag value in the subtraction.

```
* ==================================================================
*         Z-80 - 16  BIT - SUBTRACTION
* ------------------------------------------------------------------
          LD   HL,SECOND$NUMBER    ;HL points to low byte of 2nd number
          LD   A,(FIRST$NUMBER)    ;Get low byte of 1st number in Acc.
          SUB  (HL)                ;Subtract low bytes
          LD   (RESULT),A          ;Store low byte of result
          LD   A,(FIRST$NUMBER+1)  ;Get high byte of 1st number
          INC  HL                  ;Now points to hi byte of 2nd number
          SBC  A,(HL)              ;Subtract high bytes with borrow
          LD   (RESULT+1),A        ;Store high byte of result
* ==================================================================
```

A more compact version using HL and DE can also be written and again this is similar to the equivalent addition program. The only difference to be noted is that the only subtraction instruction available for the double register operations is a subtract with carry. This being so we clear the carry flag by ANDing the accumulator with itself thus producing a 'normal subtraction' (there is no explicit 'clear carry Z-80 instruction' that could be used).

```
* ==================================================================
*         Z-80 - ALTERNATIVE 16 BIT SUBTRACTION
* ------------------------------------------------------------------
          LD   DE,(FIRST$NUMBER)         ;Load DE with 1st number
          LD   HL,(SECOND$NUMBER)        ;Load HL with 2nd number
          AND  A                         ;Clear the carry flag
          SBC  HL,DE                     ;Equivalent to HL <-- HL - DE
          LD   (RESULT),HL               ;Store result
* ==================================================================
```

The above ideas can be expanded to any number of bytes and the general principles remain unchanged but for now we shall turn our attention to the slightly more complicated problem of multiplication and division. The algorithms for multiplication and division are similar and can be found from many sources. Once one algorithm is understood the understanding of the other follows quite easily. This being so we are going to restrict our examination to that of multiplication.

MULTIPLICATION

Consider the following base 10 product:

```
   25         <--- Multiplicand
   12         <--- Multiplier
 ------
  25          <--- Partial products
   50
 ------
  300         <--- Result
```

Let us take the simple product shown above and do the same calculation using base 2, i.e binary arithmetic.

```
        1 1 0 0 1        <--- Multiplicand (25)
          1 1 0 0        <--- Multiplier   (12)
 -----------------
  1 1 0 0 1              <--- Partial products
    1 1 0 0 1
      0 0 0 0 0
        0 0 0 0 0
 -----------------
1 0 0 1 0 1 1 0 0        <--- Result (300)
```

The important point is that the partial products are either zeros or a shifted version of the multiplicand. We can use this knowledge to devise an algorithm for binary multiplication. For each bit in the multiplier we must ask, 'is this bit set to 1?', if it is then we add the shifted equivalent of the multiplicand to the result. Two approaches

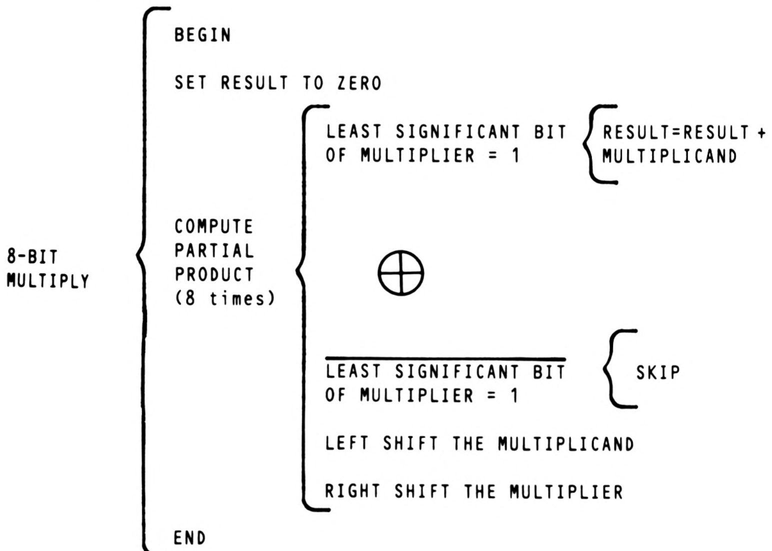
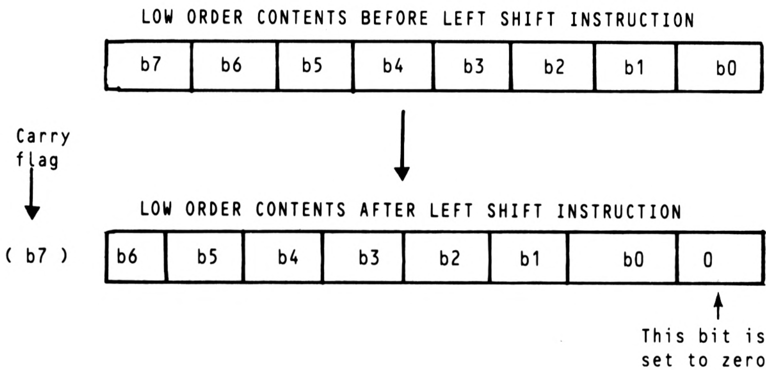


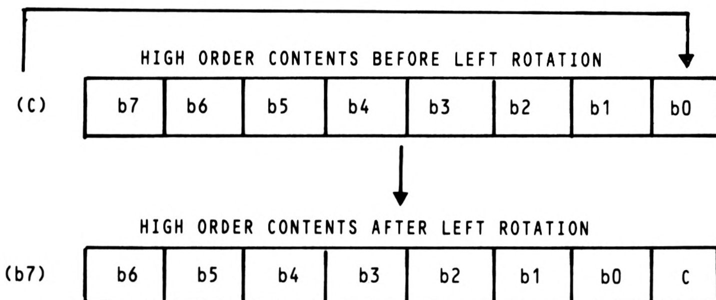
Figure 8.3: **Warnier diagram for an 8-bit multiply.**

are possible, we can either 'left shift' the multiplicand during the operations or we can 'right shift' the bytes or registers that are storing the result. The Warnier diagram for a basic 8-bit multiply algorithm is shown in figure 8.3.

Before showing some typical code for an 8-bit multiplication we need to understand the general ideas behind creating 16-bit shifts. In general the left shift operations available on our microprocessors will push bit 7 into the carry flag. When attempting to left shift a 16-bit (i.e 2 byte) value we can use a normal left shift on the low order byte as follows:



Bit 7 falls into the carry flag and to obtain a 16-bit shift we now wish to shift this bit, now in the carry flag, into bit 8 of the 16-bit number, i.e we want to push this carry value into bit 0 of the high order byte. We need an instruction that performs a left shift that includes the carry and the most commonly implemented instructions that perform this are called rotation instructions. Rotation to the left thus has the following effect.



By utilizing a combination of left shift on the low order byte, and a left rotation (through the carry) on the high order byte we can left shift a 16-bit number held in two bytes or in two 8-bit registers. The principles can of course be extended to any number of bytes as required. Similar instructions are usually available for the equivalent right shifts and right rotations. Occasionally you will find tricks being used to create 16-bit left shifts. One favourite on the Z-80 is using the double register addition instructions to add a register pair to itself: ADD HL,HL results in a 16-bit arithmetic left shift.

Let us see how these ideas help to produce a simple multiplication program that takes an 8-bit number held in a location labelled MULTIPLICAND, multiplies it by a second number held in location MULTIPLIER, and finally places the result into the two bytes starting from the lowest byte which has been labelled RESULT as shown in figure 8.4.

The following code is split into two parts. Firstly we load the registers with the following data: HL is loaded with the address of the multiplier and register C is then loaded with the multiplier itself (using indirect addressing through HL). A 'bit count' of 8 is loaded into the B register and this will be used to count how many times we have gone through the 'multiplication loop'. The HL pair are then incremented so that they then point to the multiplicand, which is

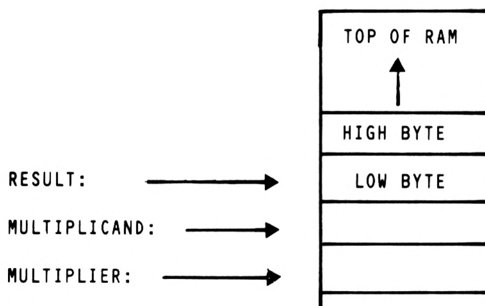


Figure 8.4: **Layout in memory of 8-bit multiplication**

placed in the E register using a LD E,(HL) instruction. Register D is set to zero because, although the multiplicand is only 8 bits we will need 16 bits available as we do the 16-bit left shift operation explained earlier. Finally HL is set to zero and will be used to collect the result prior to storing it in locations RESULT and RESULT+1.

The second section of code is the actual multiplication itself. We use a *right shift* operation on the C register so that the least significant bit goes into the carry.

This means that if the carry becomes set then the least significant bit was a 1. The carry flag is tested and if it has not been set then the partial product is zero and we skip the addition. Before moving on to the start of the loop again the DE pair is shifted using a left shift followed by a left rotation, and the 'bit counter', B, is decreased. If B is *not zero* then we repeat the loop again otherwise the final result is stored in RESULT and RESULT+1.

```
* ===================================================================
*         Z-80 EIGHT BIT MULTIPLICATION
* -------------------------------------------------------------------
          LD   HL,MULTIPLIER     ;HL points to multiplier
          LD   C,(HL)            ;Get multiplier in C register
          LD   B,8               ;B is used as a 'bit' counter
          INC  HL                ;Now HL points to multiplicand
          LD   E,(HL)            ;Get multiplicand in E register
          LD   D,0               ;Now DE = multiplicand!
          LD   HL,0              ;HL will be used to hold result
* -------------------------------------------------------------------
MULTIPLY: SRL  C            ;Least sig. bit (multiplier) into carry
          JR   NC,SKIP           ;Indicates Least sig. bit is zero
          ADD  HL,DE             ;Add partial product to result
SKIP:     SLA  E                 ;Left shift multiplicand low byte
          RL   D                 ;Left rotate high byte through carry
          DEC  B                 ;Decrease bit counter
          JP   NZ,MULTIPLY       ;Do next bit
          LD   (RESULT),HL       ;Store result
* ===================================================================
```

This 'first attempt' code can be shortened and improved in several ways. The Z-80 has a combined '*decrement and relative jump on not zero*' instruction. It operates using the B register as the counter: it decreases the B register by 1, then if B<>0 the relative jump is performed. Another improvement is also possible but is less obvious. If the multiplier is placed in the H register and the L register set to zero then the instruction ADD HL,HL will perform a 16-bit left shift. As the multiplier is shifted out during processing we create room to store the result in HL.

To take advantage of this arrangement we must shift the multiplier to the *left* to deal with the most significant partial product first. We can also tighten up the initial loading code by loading HL as a register pair starting one byte *below* the multiplier (so that the multiplier goes into the H register). The L register can be cleared after this 16-bit load ready to receive the result. A similar trick can be used to load the multiplicand into the E register.

These improvements have been made in the version shown below. Bear in mind that we have now made *processor specific* enhancements, we have not improved the basic steps of the algorithm but have, by appropriate choice of registers and a 16-bit left shifting trick, produced a better practical implementation.

```
* ==================================================================
*        Z-80 EIGHT BIT MULTIPLICATION - VERSION 2
* ------------------------------------------------------------------
          LD   HL,(MULTIPLIER-1)  ;Get multiplier in H register
          LD   L,0                ;Clear to zero
          LD   B,8                ;B is used as a 'bit' counter
          LD   DE,MULTIPLICAND    ;Get multiplicand in E register
          LD   D,0                ;Now DE = multiplicand!
* ------------------------------------------------------------------
MULTIPLY: ADD  HL,HL              ;16 bit left shift
          JR   NC,SKIP            ;Indicates most sig. bit is zero
          ADD  HL,DE              ;Add partial product to result
SKIP:     DJNZ MULTIPLY           ;Do next bit
          LD   (RESULT),HL        ;Store result
* ==================================================================
```

FINAL WORD

Before leaving the subject of number representation, did you try the left shift experiment suggested earlier? If you did you will have found that shifting a number to the left is equivalent to multiplying the number by 2. Similarly two left shifts are equivalent to multiplying by 4. In general an *n*-bit left shift will multiply the value by 2 raised to the power '*n*'. Occasionally these ideas can be used to avoid having fully fledged multiplication routines within your programs. As an example the addresses of descriptors such as those mentioned in chapter nine are easily calculated by shifting – *if* the descriptor lengths are 2, 4, or 8 bytes in length.

9 DATA STRUCTURES

The use of appropriate data structures plays a vital role in the final efficiency of a program. The use of inappropriate representations can result in programs which, in use, perform poorly. A *data structure* describes the different ways in which we can impose a 'logical' structure onto a set of data. We will start by looking at some basic *data types* and the corresponding ways in which they can be represented within a computer.

FIXED AND VARIABLE LENGTHS

Sets of alphanumeric characters, 'strings' in BASIC, are usually represented by sets of equivalent ASCII codes (the ASCII code descriptions are given in appendix C). Numeric data (chapter 8) may be represented in various ways, with unsigned and signed binary, two's complement, BCD and floating point forms being the most common. We can make the immediate distinction between representations, such as floating point numbers which have a *fixed length*, and representations such as text strings in which the data items have *variable length*. The distinction is important because it affects the way in which we handle such items within the computer (*see figure 9.1*).

REPRESENTING DATA

One simple way of representing variable length data within a computer is to use a suitably sized block of memory and *delimit* each item with a specified character (such as '*'). This simple scheme can be illustrated by considering the following list of names JOHN, ANDY, SUE, PAUL.

Such a representation has the obvious disadvantage that the list must be searched sequentially in order to locate a particular item, the situation improves when we use fixed length data. Figure 9.2 provides a typical example of this with four bytes being allocated for each item. Here we do not need to delimit each piece of data explicitly, nor do we need to search the complete list to find, say, the 3rd item. For all items in such a table we can easily calculate the address of the *n*th item as BASE + 4 times (n-1). Fixed data length table structures are simple to create and manipulate.

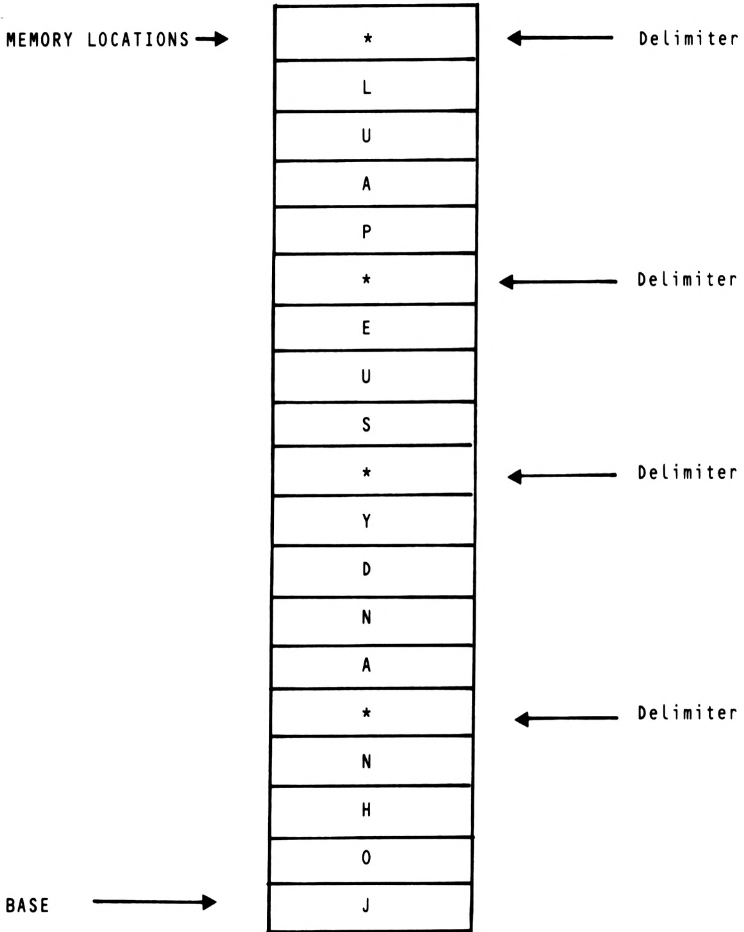


Figure 9.1: **Part of a sequential block of character data**

One disadvantage with this simple fixed data length approach is that if you try to store variable length data it will waste memory space. Another, more important, is that you must still search the whole of the table if you wish to see if an item of particular value X is present. This can be overcome by building the table in an ordered manner. If sorted tables are created they can be searched efficiently by using techniques such as '*Binary searching*'. Static tables (such as look-up tables) which are used but not altered are commonly created using table structures – they can be designed for efficient retrieval from the start.

The major problem with table structures occurs while trying to keep them in their correct order when data is added or deleted. It

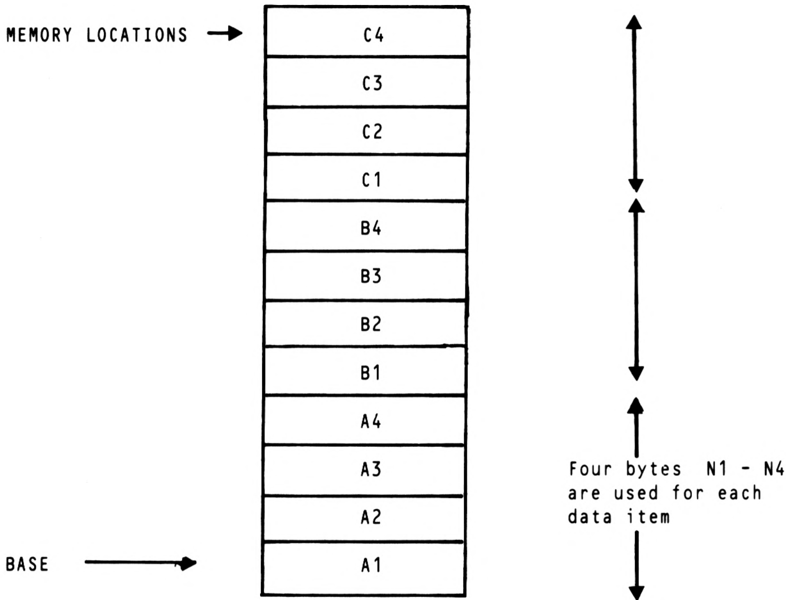


Figure 9.2: **Fixed data length items do not need delimiting**

invariably results in the need to move large blocks of data – unsatifactory in many applications where data is changed frequently. With variable length items the problems are aggravated because of the different lengths involved. In both cases as the tables increase in size any problems become ever more apparent.

STACKS

The stack is a Last In First Out (LIFO) structure and should already be familiar since the Z-80 uses one to store its return addresses. To implement a stack structure a block of memory is reserved and a 'pointer'; a two byte location or register is used to indicate the current 'top' of the stack. As data is placed on the stack, so the stack pointer is adjusted accordingly. When data is removed it is removed starting from the pointer address. The result is an effective implementation of a 'Last In First Out' structure. The fact that addition and deletion occur only at one end is a distinctive feature of the stack structure. Another distinctive feature is that the items on the stack do not, during the process of addition and deletion, get physically moved around in memory. It is of course perfectly possible to implement a LIFO structure by using a 'linked list' (see later), rather than using a contiguous block of memory as does the Z-80.

LINKED LISTS

To a large extent it is true to say that the simpler forms of representing data in blocks, tables and so on are unsuitable for representing data which is required to be kept in ordered form but which is 'dynamically changing.'

Linked lists go some way towards solving the objections in a way that is suitable for dynamically changing data. To understand what linked lists are, and how they can be created, it is necessary to understand clearly the concept of using 'pointers' to imply logical orders within data sets. Consider the four names JOHN, ANDREW, SUE and PETER in that order. We can superimpose a 'logical ordering' onto the existing physical order by the use of arrows. Figure 9.3 shows two examples.

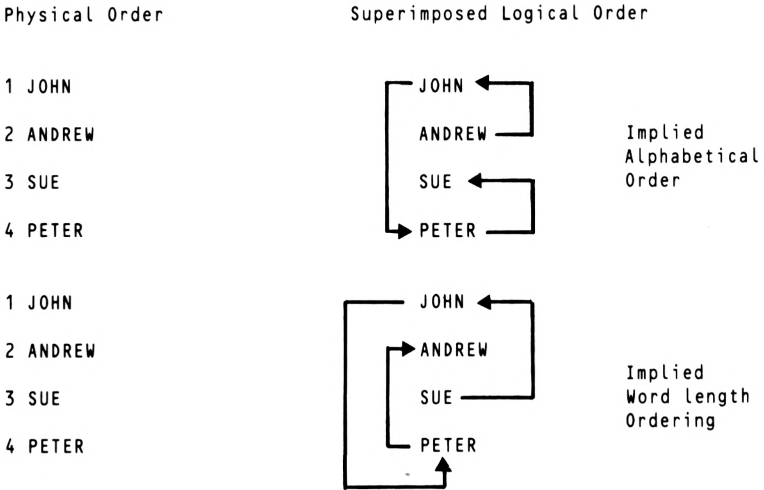


Figure 9.3: **Using 'arrows' to impose a logical ordering onto data**

Such a technique has an identical parallel in the world of computing. We can include, either within the data or external to it, a pointer to indicate the next item in the 'logical order'. In assembly language programming this pointer will most likely be an address, in higher level languages it may be a record number or the number of an element in an array.

For our purposes we can define a pointer as a two byte address from which a data item may be obtained. Usually a data item will consist of more than one byte and the most common convention is that a pointer will hold the address of the location of the first byte of the 'set'. Two types of pointers are distinquished, those that are embedded within the data, and those that are separated.

Pointers are used to implement most of the common data structures including linked lists. If we consider our earlier example of the list of names JOHN, ANDREW, SUE and PETER, then we could re-write our 'arrow' notation of an imposed alphabetical order relation as shown below.

PHYSICAL ORDER	ITEM IN MEMORY	NEXT ITEM POINTER
1. JOHN	JOHN	4
2. ANDREW	ANDREW	1 (S)
3. SUE	SUE	0 (E)
4. PETER	PETER	3

Starting from item 2, the 'head' of the chain, we can use the pointers to move throught the data items in the order Item 2→Item 1→Item 4→Item 3. We need to know the location of the start or 'head' of the chain and additionally we must provide some way of knowing when we have reached the end of the chain. In the above diagram we use (S) for the head and (E) for the end. Common practical solutions involve keeping a record of the address of the head of the chain and using a null pointer (such as a zero address) to signify the end of the chain. We can easily use the same data items as parts of more than one list simply by adding additional pointers:

		ALPHABETICAL ORDER	WORD-LENGTH ORDER
PHYSICAL ORDER	ITEM IN MEMORY	NEXT ITEM POINTER	NEXT ITEM POINTER
1. JOHN	JOHN	4	4
2. ANDREW	ANDREW	1 (S)	0 (E)
3. SUE	SUE	0 (E)	1 (S)
4. PETER	PETER	3	2

The advantages in this type of representation may be seen if we now add a further item to our list. Let us add the name CATHERINE. To do this we place the item at the end of our data list, but we adjust the pointers as explained in the following alphabetical example. Since the word CATHERINE comes before JOHN but after ANDREW (alphabetically) we must change the ANDREW pointer so that it points to CATHERINE rather than JOHN. We must then associate with CATHERINE a pointer to the item JOHN. In this way we have maintained the logical order. We can do a similar exercise with the other pointers (ie. those relating to the word length order). If we do this we obtain the following representation:

PHYSICAL ORDER	ITEM IN MEMORY	ALPHABETICAL ORDER NEXT ITEM POINTER	WORD-LENGTH ORDER NEXT ITEM POINTER
1. JOHN	JOHN	4	4
2. ANDREW	ANDREW	5 (S)	5
3. SUE	SUE	0 (E)	1 (S)
4. PETER	PETER	3	2
5. CATHERINE	CATHERINE	1	0 (E)

The name CATHERINE has been added to the end of the data, and we have maintained not just one, but both ordered lists in their correct forms without physically rearranging the data.

MULTIPLE LINKS

As well as forward pointers it is possible to use pointers which point to the preceding item. It is frequently useful to include both forward and backward links to further facilitate deletion, insertion and other functions.

CIRCULAR LISTS

Linked lists in which the last element points back to the first are often used in multiple servicing routines and certain 'hashing' techniques (key to address transformation!). It is necessary to keep track of which item is currently 'active' in such lists.

QUEUES

These are First In First Out (FIFO) structures. They can be implemented in various ways, for example by making additions to the *end* of a linked list.

TREES

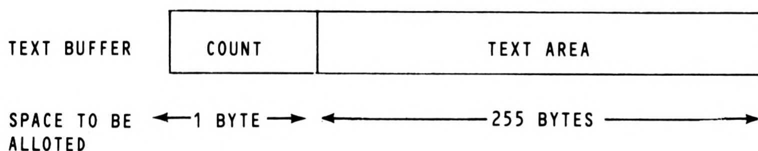
Trees are extremely important structures and enable particularly efficient *sorting* and *searching* routines to be developed. An introduction to the concept of a *tree structure* is given in chapter 10.

A WORKING MODEL

The example that has been selected gives a simple illustration of the use of 'pointers' and also helps to illustrate how a particular problem can be broken down to provide sufficient useful information to create the final assembly language code. We will collect text from the keyboard and store it in an area of memory designated as 'string space'. As we do so we will build a 'descriptor' set to tell us the size and location of each entry in memory (this information will be used to print back the list of entries when input is complete).

STRING INPUT ROUTINE

We will require an input routine to collect a string of characters from the keyboard. We will place those characters in a '*buffer*' – an area of memory in which we can temporarily place items before we transfer or use them. The buffer will have a fixed address and we shall label its start as TEXT$BUFFER in our source program. To transfer characters we will need to keep track of how many characters have been collected, so a character count will be needed. We can identify an arbitrary, but suitable, buffer format as follows:



The source program will include a reserve data space declaration such as:

```
TEXT$BUFFER:  DS  256  ;Byte 1 =Count :Bytes 2 - 256 =Data
```

Assume that input words will be typed in and followed by a carriage return. The essential structure required is shown in figure 9.4.

This is similar to structures examined in chapters 5 and 6. This type of coding can be written from the diagram and in this particular case the 'nested subroutine' arrangement is unnecessary.

Overall operation is straightforward: pointers are set up and a simple loop is used to collect the input character and store it in the buffer area. Each time a character is obtained which is not a 'carriage return' it is stored and the count increased. The C register will be used for the count and strings will be restricted to 255 bytes or less (we will initialize BC to zero, rather than just C because this will prove useful later). HL is loaded with the address of the start of the buffer and then a loop is used to collect characters. As input is collected we check to see if a carriage return has been typed. If the character is not a carriage return then it is placed into the buffer area using HL as an indirect pointer. As soon as a carriage return is detected we jump out of the loop and perform a 'close buffer' operation. This entails writing the 'count' (the contents of the C register) at the start of the buffer. Here is a starting point:

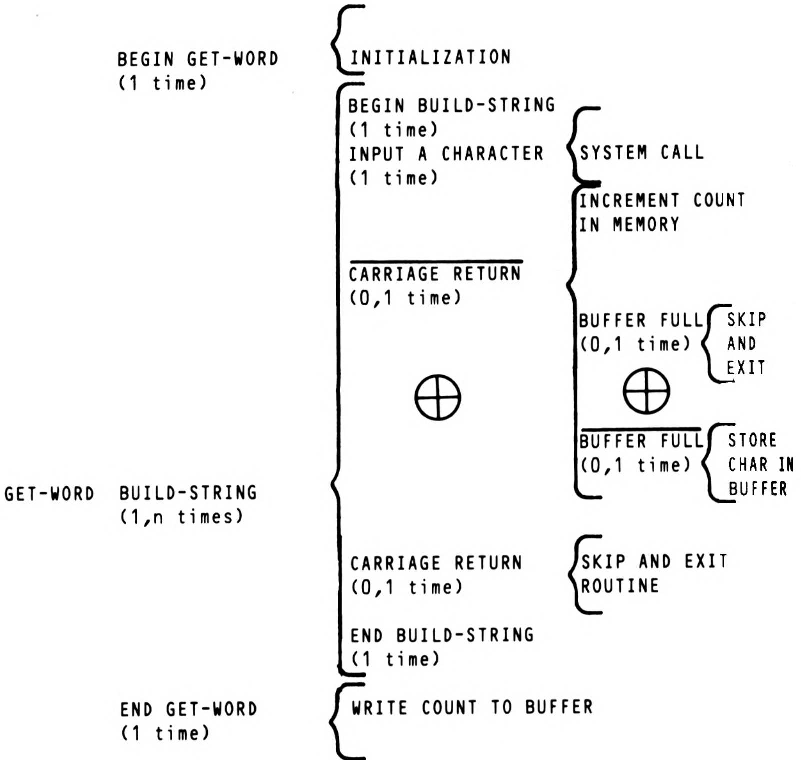


Figure 9.4: **Essential input requirements**

```
* ==================================================================
*              G E T - W O R D - S U B R O U T I N E
* ------------------------------------------------------------------
GET$WORD:       LD   BC,0                     ;Initialize count
                LD   HL,BUFFER$SPACE          ;Start of buffer
BUILD$STRING:   CALL INPUT$ROUTINE            ;System call
                CP   CARRIAGE$RETURN          ;is it a CR ?
                JR   Z,CLOSE$BUFFER
                INC  C                        ;Increase count
                INC  HL                       ;Increment pointer
                LD   (HL),A                   ;Store character
                JR   BUILD$STRING             ;Back for next character
CLOSE$BUFFER:   LD   HL,BUFFER$SPACE          ;Need start address again
                LD   (HL),C                   ;Store character count
                RET                           ;Return from routine
* ==================================================================
```

As an initial attempt, the above routine is fine except for one thing – a check has not been made to see if the input string is greater than 255 characters. The easiest solution is to check the count immediately after the INC C instruction. If C goes over 255 the zero flag will be set. The additional check is simple to implement: we use a *relative*

jump on zero instruction to branch to CLOSE$BUFFER if the zero flag has been set. Since C is incremented fromm 255 to 0 when the buffer overflow is indicated the effect is that an oversize entry will be completely ignored and treated as a 'null entry'.

The additional instruction has been added in the final version shown below. The call to CR$LF is a carriage return/line-feed routine, a subroutine to output ASCII codes 13 and 10 together. Such a routine may be available as a system call or it may be necessary for you to write a short subroutine to perform the function. Without such a routine the input words will display as a single continuous line of characters.

```
* ==================================================================
*               G E T - W O R D - S U B R O U T I N E
* ------------------------------------------------------------------
GET$WORD:       LD   BC,0                   ;Initialize count
                LD   HL,BUFFER$SPACE        ;Start of buffer
BUILD$STRING:   CALL INPUT$ROUTINE          ;System call
                CP   CARRIAGE$RETURN        ;is it a CR ?
                JR   Z,CLOSE$BUFFER
                INC  C                      ;Increase count
                JR   Z,CLOSE$BUFFER         ;Buffer is full
                INC  HL                     ;Increment pointer
                LD   (HL),A                 ;Store character
                JR   BUILD$STRING           ;Back for next character
CLOSE$BUFFER:   LD   HL,BUFFER$SPACE        ;Need start address again
                LD   (HL),C                 ;Store character count
                CALL CR$LF                  ;Output Carriage return
                                            ;and Line feed
                RET                         ;Return from routine
* ==================================================================
```

TRANSFERRING DATA

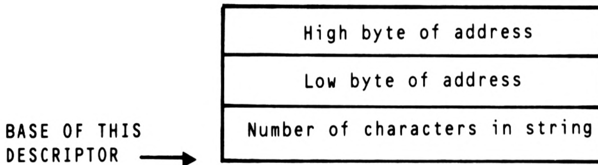
To transfer data from the buffer we need a *source address* and a *destination address*. The source address is the buffer. The destination address depends on the next byte of the string space that is available. This will vary as we add data, so we will keep track of the address in a two byte static variable that called NEXT$FREE$STRING$SPACE. In the source program we shall see a further space declaration such as:

```
NEXT$FREE$STRING$SPACE DS 2         ;Next free string space location
```

It will of course be necessary to initialize these locations to a suitable value.This may be done using instructions at the start of the program or it may be done within the source code by using a '*define bytes*' assembler directive.

A simple definition of a descriptor is that it is an external 'pointer set' identifying certain characteristics about a data item. Our descriptors will contain the size of a data item and the address of the

first byte of the item. Since we restrict the size of an any input string to 255 bytes or less, we can hold the 'count' in one byte of the descriptor. Two bytes are required for an address so each descriptor will require a total of three bytes. As always the Z-80 address convention is used, with the low byte of the address being stored first. The following illustration shows the schematic layout of the descriptor.



When placing an incoming item into string space, it is necessary to create a suitable descriptor for the item.

A special two-byte count variable is reserved to keep track of the number of descriptors created. We will call it DESCRIPTOR$COUNT and it will be defined as the last two bytes of the object program. It will therefore reside immediately below the base of the descriptor table giving certain practical advantages. For convenience a record of the descriptor size (3 in this case) will also be kept in a two byte variable called DESCRIPTOR$SIZE.

The overall layout or memory map for our example should now be becoming apparent. A typical schematic outline for our selected memory use is shown in figure 9.5

The start of the program will depend on the particular computer being used, as will the address of the start of string space – it is up to you to decide which area of memory is appropriate. The most useful general point is that these memory arrangements are essentially arbitrary. You can as easily have descriptors building down from 'high memory' and strings moving upwards from above the end of the program's static variable area.

If we assume for the present moment that a 'word' is already available in the buffer area then we can consider details of the transfer mechanism. Firstly it will be helpful if we can use one of the automated 'block transfer' instructions. We have chosen to use the decrement form of the repeating block load, LDDR, since this will simplify the transfer coding. To do this we must use HL as a source pointer, DE as a destination pointer, and BC as a two byte counter. With this arrangement in mind we define the following uses for the registers and register pairs shown in figure 9.5.

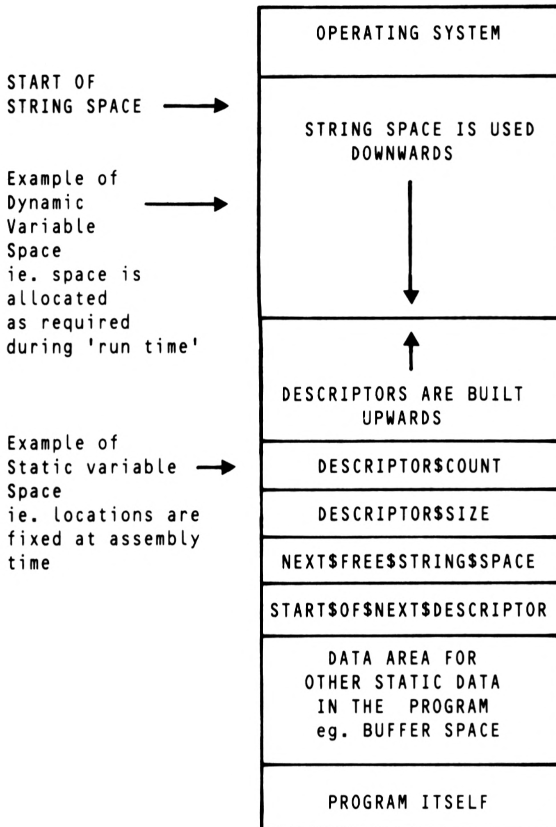


Figure 9.5: **Memory map for the example program**

BC: Register C will be used to hold the count of the number of characters in the buffer. By initializing BC to zero during the 'Get-word' subroutine we ensure that BC = C.

DE: is used as the 'destination' pointer. It will point into string space.

HL: This will be the 'source' pointer and will point into the buffer.

IX: Will point to the first free byte above the existing descriptor set. The value will come from the static variable we have called STARTOFNEXT$DESCRIPTOR. The IX pointer will be used to put data into new descriptors.

COPYING DATA INTO STRING SPACE

We are assuming here that IX and DE have been set up to point to the *base of the new descriptor* and the *next free string space* byte respectively.

The block transfer instructions provide a means of automating the movement of blocks of data. LDDR is one such instruction and we shall use it to transfer data from the buffer to the string space area. To use LDDR it is necessary to load HL with the source start address, DE with the destination start address, and BC with the number of bytes to be transferred. The instruction will transfer the byte pointed to by HL to the byte addressed by DE, it will then decrement BC, HL and DE and if BC is not equal to zero the instruction will be performed again. This repetition will continue until BC becomes equal to zero.

On leaving the 'Get-word' subroutine, HL and BC contain the start address of the buffer and the character count, consequently an ADD HL,BC instruction will 'point' HL to the last character placed in the buffer. This sets up HL as the source pointer for an automated transfer. At this point we have to place the character count into the first byte of the descriptor because otherwise the LDDR transfer will destroy the count value. After the block transfer DE will be pointing to the byte below the last byte of string space used, this is used to update NEXT$FREE$STRING$SPACE. Incrementing DE results in DE pointing to the first character of the new word in string space and this 'start address' is then stored in the new descriptor using indexed addressing with code like this:

```
ADD  HL,BC                          ;Now points to last character
LD   (IX+0),C                       ;Count in descriptors first byte
LDDR                                ;Perform block transfer
LD   (NEXT$FREE$STRING$SPACE),DE    ;Update with new value
INC  DE                             ;Now points to start of new word
LD   (IX+1),E              ;Store low byte of new string's address
LD   (IX+2),D                       ;Store high byte of address
```

Notice that because LDDR decreases HL and DE as it re-executes, the transfer process is copying the buffer word starting from the end of the word. Figure 9.6 illustrates this.

UPDATING THE STATIC VARIABLES

By re-using the DE register pair the descriptor size is added to the contents of index register IX (so that it then points to the next byte above the newest descriptor) and the result stored in STARTOFNEXT$DESCRIPTOR. Finally the descriptor count is updated to reflect the fact that a new descriptor has been created. Typical coding is shown below:

```
LD  DE,(DESCRIPTOR$SIZE)              ;Get descriptor size in DE
ADD  IX,DE                             ;Add to current descriptor base
LD   (START$OF$NEXT$DESCRIPTOR),IX     ;and store new base
LD   HL,(DESCRIPTOR$COUNT)             ;Get count
INC  HL                                ;Increment
LD   (DESCRIPTOR$COUNT),HL             ;Replace count
```

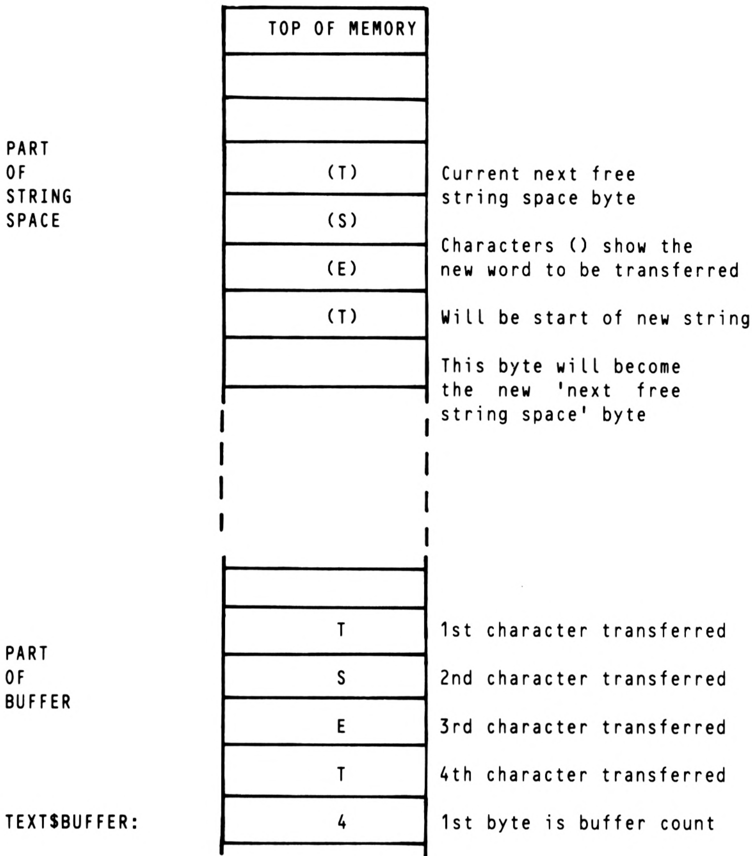


Figure 9.6: **Example of transfer from buffer to string space**

Before completing a finished subroutine we must ask what will happen if so much data is entered that the descriptors being built *upwards* meet the end of the string string space that is moving *downwards*?

At present we have not identified any check that could prevent this major disaster from happening. It is helpful to look at a Warnier diagram of the 'store data' routine so that we can be sure of the objectives to be set. Figure 9.7 gives the general layout that is required.

We must decide what conditions determine whether the new data is 'safe to write'. Prior to the LDDR instruction being executed, DE is pointing to the current next free string space byte and BC is the character count.

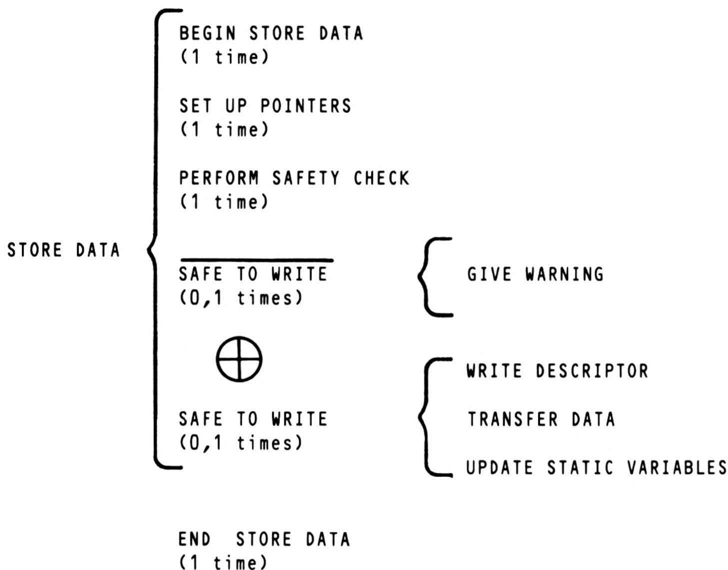


***Figure 9.7:* Layout of the 'store data' subroutine**

DE–BC+1 is therefore the address of the lowest string space byte that would be used if the new data item was written. Similarly the highest descriptor byte that would be used would be IX+2. The condition for 'safe' storage is therefore given by the solution set of the inequality DE–BC+1 > IX+2. This can be written in the form DE–BC–1 > IX from which we obtain IX+BC+1–DE < 0. To perform this check we add BC to IX and then increment IX, this produces the equivalent of IX+BC+1 in index register IX. By transferring IX to HL and using SBC HL,DE we effectively provide the required function. If the carry flag has been set then it *is* safe to store the new data. In the code that follows notice how the stack is used to copy the IX register into the HL pair providing the function HL←IX.

```
* =================================================================
*            S A F E  - T O  -  W R I T E  -  C H E C K
* -----------------------------------------------------------------
CHECK:          PUSH IX ! PUSH HL         ;Preserve registers
                ADD  IX,BC                ;Will also clear carry
                INC  IX                   ;Gives IX <--- IX+BC+1
                PUSH IX
                POP  HL                   ;Copy trick
                SBC  HL,DE                ;Carry set if safe
                POP HL ! POP IX           ;Restore registers
                RET                       ;Return from subroutine
* =================================================================
```

To implement a 'warning' we could output a 'bell character' (ASCII 7), and a routine to do this is shown below:

```
* ==================================================================
*        U N S A F E  - T O  -  W R I T E  -  W A R N I N G
* ------------------------------------------------------------------
WARNING:       LD   A,BELL                     ;Signifies  no  space
               CALL OUTPUT$ROUTINE             ;available for further
               RET                             :input data
* ==================================================================
```

We are now in a position to link the various sections of code together to produce a completed version of the 'store data' subroutine. The code has been produced by using the guideline structure given by the Warnier diagram, coupled with the individual section analysis of how to implement the various functions needed.

```
* ==================================================================
*            S T O R E - D A T A - S U B R O U T I N E
* ------------------------------------------------------------------
STORE$DATA:  PUSH AF                           ;Preserve flags/acc
             LD   DE,(NEXT$FREE$STRING$SPACE)
             LD   IX,(START$OF$NEXT$DESCRIPTOR) ;Start of descriptor
             CALL CHECK                        ;Any space left ?
             CALL NC, WARNING                  ;Do not store
             CALL C,WRITE                      ;Store the data
             POP  AF                           ;Restore flag/acc
             RET                            ;Logical end of subroutine
* ==================================================================
*            S A F E  - T O  -  W R I T E  -  C H E C K
* ------------------------------------------------------------------
CHECK:       PUSH IX ! PUSH HL              ;Preserve registers
             ADD  IX,BC                     ;Will clear carry
             INC  IX                        ;Gives IX <--- IX+BC+1
             PUSH IX
             POP  HL                        ;Copy trick
             SBC  HL,DE                     ;Carry set if safe
             POP  HL ! POP IX               ;Restore registers
             RET                            ;Return from subroutine
* ==================================================================
*                      W R I T E - D A T A
* ------------------------------------------------------------------
WRITE:       ADD  HL,BC                ;Now points to last character
             LD   (IX+0),C             ;  Count in desc's first byte
             LDDR                           ;Perform block transfer
             LD   (NEXT$FREE$STRING$SPACE),DE ;Update with new value
             INC  DE                    ;Points to start of new word
             LD   (IX+1),E             ;Store lo byte new strings add
             LD   (IX+2),D                 ;Store hi byte of address
             LD   DE,(DESCRIPTOR$SIZE)    ;Get descriptor size in DE
             ADD  IX,DE               ;Add to current descriptor base
             LD   (START$OF$NEXT$DESCRIPTOR),IX  ;and store new base
             LD   HL,(DESCRIPTOR$COUNT)        ;Get descriptor count
             INC  HL                    ;Increment descriptor count
             LD   (DESCRIPTOR$COUNT),HL             ;Store new count
             RET                            ;Return from subroutine
```

```
* ================================================================
*           U N S A F E - T O - W R I T E - W A R N I N G
* ----------------------------------------------------------------
WARNING:    LD   A,BELL                    ;Signifies  no  space
            CALL OUTPUT$ROUTINE            ;available for further
            RET                            ;input data
* ================================================================
```

PRINTING THE WORD LIST

The objectives are straightforward. Words are available for printing as long as the 'number of words printed' is less than the DESCRIPTOR$COUNT. In general the starting address and character count of the *n*th word is obtained from the *n*th descriptor. A routine 'PRINT-WORD' is needed to print *n* characters starting at address pq. Both the count and the starting address are easily obtained so we require a simple loop to produce the desired effect. The main loop is fairly obvious and uses the combined '*decrement and relative jump on not zero*' instruction DJNZ (this operates using the B register as the counter). Typical coding is shown below:

```
PRINT$WORD: LD   A,(HL)          ;Get character
            CALL OUTPUT$ROUTINE  ;System call
            INC  HL              ;Increment pointer
            DJNZ PRINT$WORD      ;Back for next character if B<>0
```

We have to decide how to initialize the HL and B values and we have to decide under what circumstances the routine should be called. If we load IX with the address of the base of a descriptor then we can load the necessary data into B, H and L registers using indexed addressing as follows:

```
LD   B,(IX+0)       ;Load count from descriptor
LD   L,(IX+1)       ;Low byte of address
LD   H,(IX+2)       ;Load high byte of address
```

The two byte descriptor size variable is used only for the convenience of being able to use the paired register addition instructions. In practice the size resides in the low byte only. We are therefore able to load BC initially with the descriptor size then overwrite the B register with the word character count whilst leaving the descriptor size still present in register C. To move from one descriptor to another we temporarily utilize the accumulator to save the contents of the B register then set B to zero so that an ADD IX,BC instruction produces the required function IX←IX+C. The code below gives the general idea:

```
LD   A,B            ;Preserve B
LD   B,0            ;Now BC = C
ADD  IX,BC          ;IX <--- IX + C
LD   B,A            ;Restore B
```

The only remaining problem is how to decide whether words are available for printing. There is a simple solution, we load DE with the descriptor count and decrease it by one each time a 'word' is printed. When DE is reduced to zero then all words will have been printed. Because the instruction DEC DE will not affect the zero flag it is necessary to test for DE = 0 in the following manner: the accumulator is loaded from the E register and then an OR D instruction is used. If D = E = 0 (ie. if DE = 0) then the zero flag will be set as required. Since we must check to see whether any words have been entered at all, we use a *pre*-test – we check DE on *entry* to the loop rather than at the end of the loop.

The above ideas can be combined to form the subroutine PRINT. After each word is printed we make a subroutine call to a routine called CR$LF to output a carriage return character and then a line-feed character. Notice that because the descriptor set starts *immediately* above the descriptor count variable we can use DESCRIPTOR$COUNT+2 to identify its location. In this example we could just as well have used an additional label but, in cases where the descriptor space (including the starting address of the descriptor block!) may be allocated dynamically and addressed indirectly, it is useful to be able to access the count, the descriptor size and the descriptor set itself from simple functions of just one indirect pointer.

```
* ==================================================================
*                P R I N T - S U B R O U T I N E
* ------------------------------------------------------------------
PRINT:          LD   BC,(DESCRIPTOR$SIZE)     ;Size is really in C
                LD   DE,(DESCRIPTOR$COUNT)    ;Descriptor count
                LD   IX,DESCRIPTOR$COUNT+2   ;Base of first descriptor
PRINT$1:        LD   A,E                     ;Low byte of descriptor count
                OR   D                       ;Zero flag set if D=E=0
                RET  Z                       ;Logical end of routine
                LD   B,(IX+0)                ;Load count from descriptor
                LD   L,(IX+1)                ;Load low byte of address
                LD   H,(IX+2)                ;Load high byte of address
                LD   A,B                     ;Preserve B
                LD   B,0                     ;Now BC = C
                ADD  IX,BC               ;Now IX points to next descriptor
                LD   B,A                     ;Restore B
PRINT$WORD:     LD   A,(HL)                  ;Get character
                CALL OUTPUT$ROUTINE          ;System call
                INC  HL                      ;Increment pointer
                DJNZ PRINT$WORD           ;Back for next character if B<>0
                CALL CR$LF                   ;Carriage return/line-feed
                DEC  DE                      ;Decrease descriptor count
                JR   PRINT$1                 ;Now do next word
* ==================================================================
```

PUTTING THE PIECES TOGETHER

Having now developed three subroutines to collect, store and re-display the input data, it is time to link them together. We collect 'words' from the keyboard until such time as a 'null word', ie. just a carriage return, is entered. Each word entered is stored using the 'STORE DATA' subroutine. When the end of input is detected a PRINT DATA subroutine is called to read back the input words. Look at the Warnier diagram in figure 9.8.

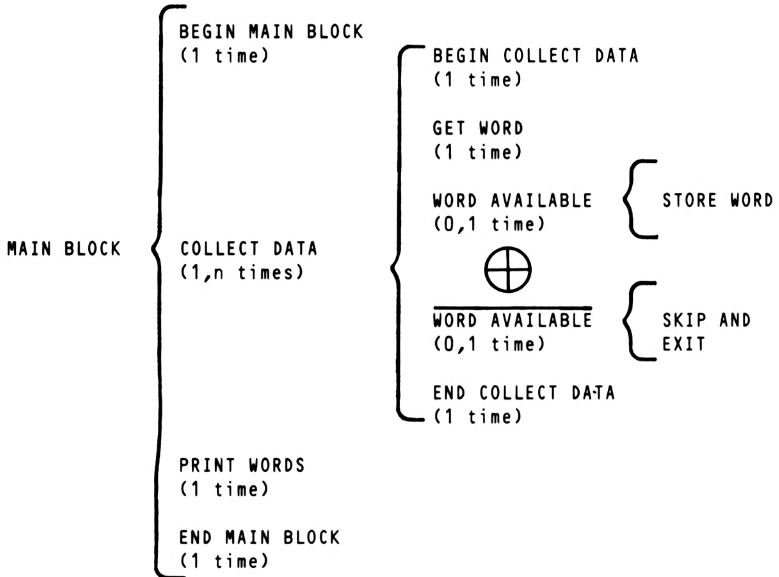


***Figure 9.8:* Structure needed to link the three subroutines**

How do we tell if a word is 'available'? We could look at the character count in the buffer, but by looking at the 'Get Word' subroutine we can see that the count is present in the C register already. By transferring this count to the accumulator it is possible to use a comparison instruction to see if it is zero. If it is then we simply call the 'Print' routine to finish, otherwise we store the data, and then branch back for further input.

A typical code is shown below. The only point that should be made is that since we use a zero flag test in two consecutive instructions we must protect it from alteration. On occasions like this the procedure used is to PUSH the combined accumulator-status flag register pair onto the stack at the start of the subroutine, and POP them off before returning. It was for this reason that PUSH/POP instructions were included in the final 'store data' subroutine.

```
* ==============================================================
*                 M A I N - B L O C K
* --------------------------------------------------------------
MAIN$BLOCK:    CALL GET$WORD          ;Collect input word
               LD   A,C               ;Copy count into accumulator
               CP   0                 ;Is it zero ?
               CALL NZ,STORE$DATA
               JR   NZ,MAIN$BLOCK     ;Get next word
               CALL PRINT
               JP   0                 ;Re-boot operating system
* ==============================================================
```

All that is needed now is some initial coding to set up the stack and to perform initialization of variables, and some data declaration statements so that we can create a 'runnable' program. As always, the exact form is dependent on your particular system, but the basic source code layout will take a form much like that which we now describe.

SOURCE CODE LAYOUT

Your program will start with an 'initialization block', which will set up the stack and define any EQUates. This will be followed by the 'main block' code, the three subroutines plus any additional routines which may be needed (perhaps to produce a combined carriage return and line feed). Finally the space reservation statements are made. The facilities offered by different assemblers vary, but most will allow you to define uninitialized space, initialized space (single bytes with a specified value), and also double bytes (often called 'words') of a specified value. Typical examples are:

```
LABEL:  DS 256   ; The assembler skips over 256 bytes
LABEL:  DB   3   ; The assembler places the value 3  into the  byte
LABEL: DW FF30H  ; The assembler  places  30 hex  into  this  byte
                 ; and FF hex into  the  byte   LABEL+1.
                 ; ie. it treats the value as a  two byte address
                 ; and stores  the low byte followed by the high byte
                 ; as is the usual Z-80 convention
```

Your assembler may use different terminology but the overall layout of the final source code before assembly should take the form shown in figure 9.9.

```
                    TYPICAL SOURCE CODE LAYOUT
SET UP BLOCK
MAIN BLOCK
GET WORD SUBROUTINE
STORE DATA SUBROUTINE
PRINT SUBROUTINE
ANY OTHER SUBROUTINE REQUIRED (eg. for CR/LF or I/O wait loops)
BUFFER$SPACE:                 DS   256
START$OF$NEXT$DESCRIPTOR:     DW   PROGRAM$FINISH
NEXT$FREE$STRING$SPACE:       DW   CB00H      ;Depends on your system
DESCRIPTOR$SIZE:              DW   3
DESCRIPTOR$COUNT:             DW   0          ;Initialized at assembly
PROGRAM$FINISH:               DS   1          ;Dummy end for label use
```

Figure 9.9: **Typical source code layout for the example**

FINAL WORD

We have tried to keep the routines relatively straightforward and have not particularly tried to minimize the code size. Some ideas of general interest have been incorporated and one example of this is seen in the 'Get-word' routine. 'Get-word', as you will remember, closes the buffer by writing the character count to the head of the buffer. For this particular example we do not actually use this stored count because it is already present in the C register when we enter the 'Store-data' subroutine. On other occasions, when other processing occurs between the time that the buffer is written and the time it is actually used, the stored count might well be needed.

By considering the general themes, as opposed to the actual coding, it should be possible to adapt many of the ideas discussed for use in your own projects.

10
SORTING and SEARCHING

SEQUENTIAL SEARCHING

Sequential searching, in which a block of memory is searched for a particular character or group of characters, is one of the simplest search techniques to implement.

For a search of 255 or fewer characters we can use a simple loop like this:

```
START:    LD   A, CHARACTER      ;Test character in accumulator
          LD   B, BLOCK$SIZE     ;Number of characters
          LD   HL,BLOCK$START
LOOP:     CP   (HL)              ;Compare counts of accum'r and HL
          JR   Z, PRESENT        ;Z flag set = character found
          INC  HL                ;Move to next byte
          DJNZ LOOP              ;Keep going
PRESENT:  etc ....
```

The character to be tested for is placed in the accumulator and HL is used as an indirect pointer to search the block using a simple loop. We use B as a counter in order to make use of the *automated decrement and relative jump* instruction DJNZ.

If we try to implement a similar kind of loop with more than 255 bytes, a problem occurs – see if you can work out why the following code will not function:

```
* ==================================================================
*       FAULTY SEQUENTIAL SEARCH FOR SPECIFIED CHARACTER
* ------------------------------------------------------------------
*    Entry conditions:     HL = Start of Block
*                          BC = Block Size
*                          A  = Character to be searched for
*
*    Exit conditions:      Zero flag set = Character found at loc'n HL
*                          -------------
*                          Zero flag set = Character not found
* ------------------------------------------------------------------
START:              CP   (HL)          ;Flag modified by A - (HL)
                    RET  Z             ;Z set = Character not found
                    INC  HL            ;Move to next byte
                    DEC  BC            ;Decrease counter
                    JR   NZ, START     ;Keep going
                    INC  BC            ;Clear zero flag
                    RET                ;Before returning
* ==================================================================
```

The above code suffers from a peculiarity of the Z-80: unlike equivalent8-bit decrement instructions (such as DEC B), the 16-bit instructions such as DEC BC do not have any effect on the flag register. Because of this the loop will fail to terminate correctly when BC becomes equal to zero! It is possible to test whether BC=0 by placing, say, B in the accumulator and ORing it with C, if B=C=0 the zero flag will have been set. In order to accomplish this it is necessary to re-arrange the use of the registers if we are to avoid overwriting the 'test character' which is held in the accumulator during the loop.

On the Z-80 there is a better alternative using one of the *automated comparison instructions* to good effect. CPIR is the *Block Compare with Increment* instruction and its effect is as follows: the contents of the byte addressed by HL is subtracted from the accumulator (as with a normal comparison instruction), then HL is incremented and BC is decremented. If BC *does not equal* zero and if the byte tested did *not equal* the contents of the accumulator the instruction is automatically re-executed. On exit the zero flag is set unless the character was not found. The above routine, re-written to use CPIR now looks like this:

```
* ====================================================================
*        AUTOMATED SEQUENTIAL SEARCH FOR SPECIFIED CHARACTER
* -------------------------------------------------------------------
*    Entry conditions:    HL = Start of Block
*                         BC = Block Size
*                         A  = Character to be searched for
*
*    Exit conditions:     Zero flag set = Character found at loc'n HL
*                         -------------
*                         Zero flag set = Character not found
* -------------------------------------------------------------------
START:              CPIR                ;Automated block comparison
                    RET
* ====================================================================
```

The automated block compare instructions CPIR and the decremental form CPDR are good examples of the improvement of the Z-80 over its predecessors the 8080 and 8085.

MULTI-BYTE COMPARISONS

The comparison of two blocks of equal length can be achieved by a straightforward search, byte by byte, until a difference is found. A more interesting problem concerns the comparison of items of differing lengths such as two text strings. We shall first consider the general problem using a Warnier diagram. In this case we have opted for a more symbolic representation based on the definition of HL–B and DE–C descriptor sets which define the two words. HL and DE are the starting addresses of the words, while B and C are the

respective character counts. (When we write (HL) we are specifying the *contents* of the byte whose address is held in HL; similarly (DE) specifies the *contents* of the byte whose address is in the DE pair).

The routine will compare two words, byte by byte, until a difference is found or until it runs out of characters in one of the words. We adopt the following convention:

- If the HL–B set is alphabetically 'greatest' (based on the ASCII values of the characters), the carry flag will be cleared.
- If the HL–B set is equal to the DE–C set then the zero flag will be set, otherwise it will be cleared.

As you look at figure 10.1 remember that B and C are being decremented each time we perform the 'Check character' routine, and that HL and DE are *pointers* indicating which characters within the words are being tested.

The following, initial example has been written directly from the Warnier diagram and uses the nested subroutine arrangement mentioned in chapter 6:

```
* ==================================================================
*          F I R S T - C O D E - C O M P A R E - W O R D S
* ------------------------------------------------------------------
COMPARE$WORDS:     LD   A, 0
                   CP   B                   ;Is B zero?
                   CALL Z, B$ZERO
                   CALL NZ, B$NOT$ZERO

                        ; an exit test here followed by
                        ; a conditional jump to start of
                        ; 'COMPARE$WORDS

                   RET                     ;Logical end of subroutine
* ------------------------------------------------------------------
B$ZERO:            CP   C                  ;is C zero?
                   CALL Z, WORDS$EQUAL
                   CALL NZ, DEC$GREATEST
                   RET
* ------------------------------------------------------------------
B$NOT$ZERO:        CP   C                  ;is C zero?
                   CALL Z, HLB$GREATEST
                   CALL NZ, CHECK$CHARACTERS
                   RET
* ------------------------------------------------------------------
CHECK$CHARACTERS:  LD   A, (DE)            ;Get byte from DE-C set
                   CP   (HL)               ;compare with HL-B set byte
                   CALL C, HLB$GREATEST
                   CALL NC, TEST$FOR$EQUALITY
                   RET
```

```
* ----------------------------------------------------------------
TESTSFORSEQUALITY:  CALL NZ, DEC$GREATEST
                    CALL Z, ADJUST$POINTERS
                    RET
* ----------------------------------------------------------------
ADJUST$POINTERS:     INC  HL
                     INC  DE
                     DEC  B
                     DEC  C
                     RET
* ================================================================
```

We have not yet decided how we will pass the DEC$GREATEST, HLB$GREATEST and WORDS$EQUAL information up through the nested subroutine levels. Neither have we determined what exit condition will be used to leave the 'compare-words' subroutine.

The next scheme avoids the need to do this at all and it is based on the recognition in the Warnier diagram that all but one of the lower level operations result in an 'exit the routine' condition. If the nested calls relating to those conditions are replaced by jumps to one of the three possible endings then we do not need to make the high level exit test because we will not fall back to that loop once an exit condition is found. Notice also that the CHECK$CHARACTER section does not need to be called as a subroutine because we fall through to it automatically if the zero test directly above it should fail. Exiting through jumps and relative jumps needs a certain amount of care because it occurs after an internal subroutine call has been made (in the initial loop). It is necessary to POP this address off the stack so that the final RET instruction causes us to leave the 'compare-words' subroutine in the proper manner.

```
* ================================================================
*        C O M P A R E - W O R D S - S U B R O U T I N E
* ----------------------------------------------------------------
*    Entry conditions:    DE-C and HL-B descriptors to be set up
*    Exit conditions:     If HL-B > DE-C then carry flag is set
*                         If HL-B = DE-C then zero flag is set
* ----------------------------------------------------------------
COMPARE$WORDS:     LD   A, 0
                   CP   B                  ;Does B = 0 ?
                   CALL Z, B$ZERO
                   CALL NZ, B$NOT$ZERO
                   JR   COMPARE$WORDS      ;Do next characters
* ----------------------------------------------------------------
B$ZERO:            CP   C                  ;Does C = 0 ?
                   JR   Z, WORDS$EQUAL     ;Exit condition
                   JR   DEC$GREATEST       ;Exit condition
* ----------------------------------------------------------------
B$NOT$ZERO:        CP   C                  ;Does C = 0 ?
                   JP   Z, HLB$GREATEST    ;Exit condition
```

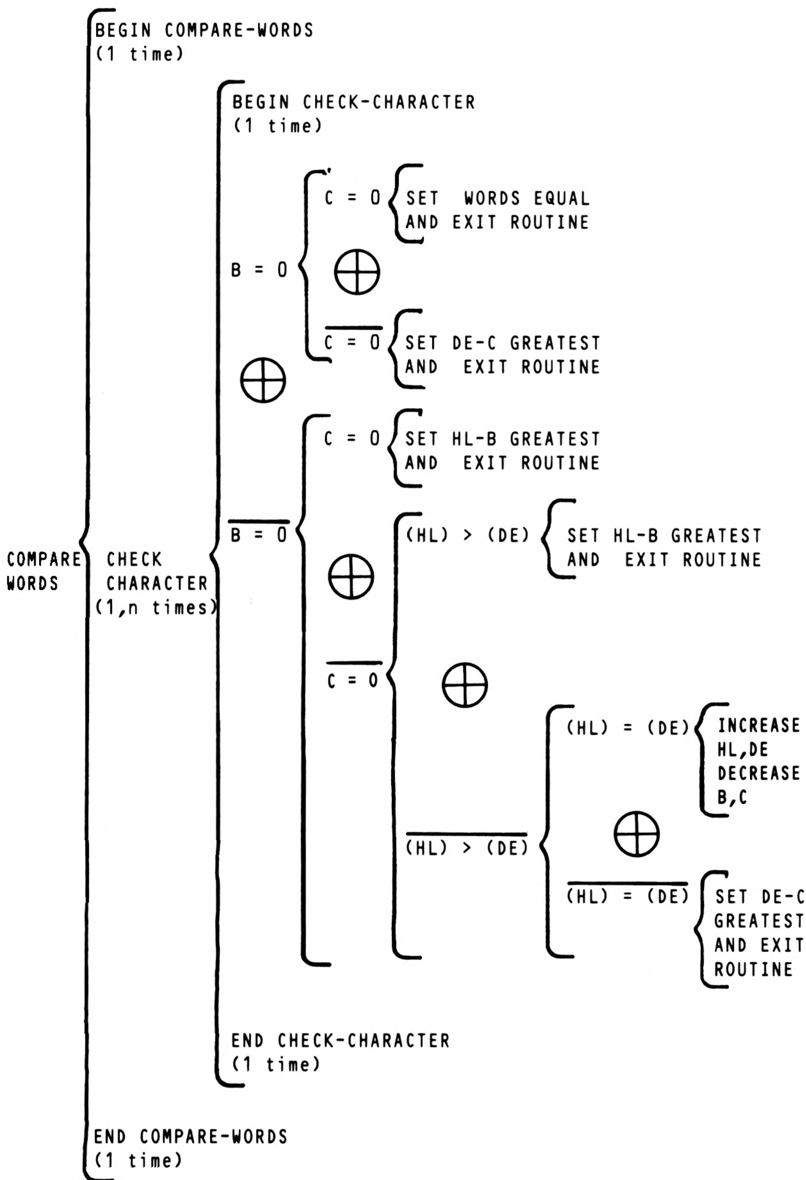


Figure 10.1: **'Compare words' routine in Warnier form**

```
* ------------------------------------------------------------------
CHECK$CHARACTER:   LD   A, (DE)
                   CP   (HL)              ;compare bytes
                   JR   C, HLB$GREATEST   ;Exit condition
                   JR   NZ, DEC$GREATEST  ;Exit condition
                   INC  HL                ;Next byte of HL-B set
                   INC  DE                ;Next byte of DE-C set
                   DEC  B                 ;Decrease B count
                   DEC  C                 ;Decrease C count
                   RET
* ------------------------------------------------------------------
WORDS$EQUAL:       POP  BC                ;Lose last call address
                   RET                    ;Leave COMPARE$WORDS
* ------------------------------------------------------------------
DEC$GREATEST:      POP  BC                ;Lose last call address
                   OR   A                 ;Clear carry flag
                   RET                    ;Leave COMPARE$WORDS
* ------------------------------------------------------------------
HLB$GREATEST:      POP  BC                ;Lose last call address
                   SCF                    ;Set carry flag
                   RET                    :Leave COMPARE$WORDS
* ==================================================================
```

Further improvements can be made by recognizing that the 'words equal' condition does not entail any action other than leaving the routine via POP and RETurn. By slightly re-arranging the end coding we can eliminate some duplicate instructions. The final version that follows includes the modified 'end' and additionally uses POP and PUSH instructions to preserve the original contents of BC, DE and HL. The routine thus makes the comparison without disturbing the HL–B and DE–C descriptor sets passing the result back using the carry and zero flags.

```
* ==================================================================
*    C O M P A R E - W O R D S - F I N A L - S U B R O U T I N E
* ------------------------------------------------------------------
*    Entry conditions:    DE-C and HL-B descriptors to be set up
*
*    Exit conditions:     If HL-B > DE-C then carry flag is set
*                         If HL-B = DE-C then  zero flag is set
*                         Accumulator contents destroyed
* ------------------------------------------------------------------
COMPARE$WORDS: PUSH BC ! PUSH DE ! PUSH HL   ;Preserve registers
                   LD   A, 0
                   CP   B                 ;Does B = 0 ?
                   CALL Z, B$ZERO
                   CALL NZ, B$NOT$ZERO
                   JR   COMPARE$WORDS     ;Do next characters
* ------------------------------------------------------------------
B$ZERO:            CP   C                 ;Does C = 0 ?
                   JR   Z, LEAVE$ROUTINE  ;Words = exit condition
                   JR   DEC$GREATEST      ;Exit condition
* ------------------------------------------------------------------
B$NOT$ZERO:        CP   C                 ;Does C = 0?
                   JR   Z, HLB$GREATEST   ;Exit condition
```

```
* ------------------------------------------------------------------
CHECK$CHARACTER     LD   A, (DE)
                    CP   (HL)                ;Compare bytes
                    JR   C, HLB$GREATEST     ;Exit condition
                    JR   NZ, DEC$GREATEST    ;Exit condition
                    INC  HL                  ;Next byte of HL-B set
                    INC  DE                  ;Next byte of DE-C set
                    DEC  B                   ;Decrease B count
                    DEC  C                   ;Decrease C count
                    RET
* ------------------------------------------------------------------
DEC$GREATEST:       OR   A                   ;Clear carry flag
                    JR   LEAVE$ROUTINE
* ------------------------------------------------------------------
HLB$GREATEST:       LD   A, 0                ;Must force reset
                    INC  A                   ;of the zero flag
                    SCF                      ;Set carry flag
LEAVE$ROUTINE:      POP  BC                  ;Lose last call address
                    POP HL ! POP DE ! POP BC ;Restore orig'l contents
                    RET                      ;and leave routine
* ==================================================================
```

In certain cases it is worthwhile checking the count values B and C before entry and 'swapping' so that, for example, the HL–B set is never shorter than the DE–C set. When this approach is used some of the comparisons within the routine can be eliminated. It is however usually necessary to re-instate the original descriptors after comparison or include some indication that a descriptor switch has occurred.

In chapter 9 we developed a simple descriptor based string storage program. A comparison routine such as that just developed provides an easy way to compare words pointed to by a pair of descriptors. The descriptor data is loaded into the respective registers to create the necessary HL–B and DE–C sets and the comparison routine called. By examining the zero and carry flag afterwards, it is possible to tell which word is the greatest.

SORTING

Within weeks of learning BASIC most of you had learned something of the mysterious 'Bubble Sort'. Performed on 20 to 30 items in memory, such sorts are quite impressive – especially if you have never seen a sort program working before. When the same technique is applied to two or three hundred – or even thousand! – items, something goes sadly amiss. For sorting anything other than small amounts of data the bubble sort written in BASIC is quite simply useless. The fault lies in the mechanism of the bubble sort itself and has little to do with the language used to implement it.

When assembly language is used something magical happens to the bubble sort, it achieves a 'respectability' of performance which suggests that some metamorphosis has occurred to the essential

algorithm. The fact of the matter is simple; the *speed* of assembler will hide the inefficiency of the bubble sort for a little bit longer before the same old problems occur. This being so we are not going to waste your time explaining in detail how to program one of the worst sorting algorithms ever devised!

Instead we are going to look at a sort technique which has, for no particularly good reason, gained a reputation for being frighteningly complex – the *Tree Sort*. By examining the fundamental ideas and relating tree sorts to programs written in easy languages such as BASIC, it is possible to develop an understanding of the general principles involved.

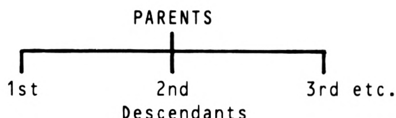
SORT TREES

All of you will have come across the term 'data structure'. The ways in which we structure our data can make a dramatic difference to the efficiency and speed with which some applications programs will run. *Stacks*, *Lists*, *Tables* and *Arrays* are examples of structures commonly used by programmers (and novices) who soon acquire a a 'mental picture' of these concepts and quickly develop proficiency in their use.

Tree structures are a very widely used way of describing and organising data, and they have many applications. The study of trees is often ignored in many popular publications, often on the ground that it is too complicated. In fact, the opposite is often true and trees can be used to radically *simplify* your problems!

Rather than simply listing various tree routines, we have decided to tackle the problem at source. We will take a guided tour through the common first approach to tree work so that you can get to grips with the underlying basic concepts before starting on the routines themselves.

You already know what a 'family tree' looks like, you know that – by convention – they are drawn 'upside-down' with parents shown *above* their descendants (The 'root' of the tree is at its *top*). Figure 10.2 illustrates this:



(This is part of a tree)

***Figure 10.2:* Tree representation**

When we draw a family tree, we are describing the relations between the parents, their children and their children's descendants – and so on. The important point is that the tree shows how these elemnts are *related* and, in the computing sense, the tree data structure is somewhat similar – and the terminology used will often reflect this.

Another common 'non-computing' example of a tree structure is the management organisation chart of which figure 10.3 is a typical example. Again the purpose is to show relations, this time between the various jobs or orders of responsibility in the company.

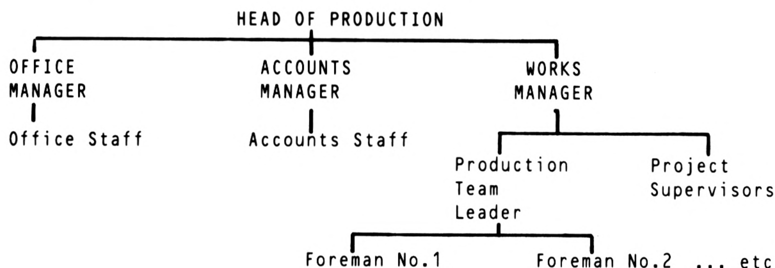


Figure 10.3: **A company hierarchy tree**

Notice that each item on the chart in Figure 10.3 is related to only *one* item above it. We say that each item has only *one parent*; Foreman No.1 is responsible *only* to the Production Team Leader. Similarly the Production Team Leader is *only* responsible to the Works Manager.

There is no such restriction on the number of 'descendants' an item may have. The Production Team Leader has many Foremen who have to report directly to him.

In general then, for a structure to be classed as a Tree Structure:

- Each item must have only one Parent
- Each item may have none, one or many 'descendants'

The exception to this is the very first item in the tree which will have descendants but no parent:

- The first item in a tree is given a special name: the *Root* of the tree.

Before we start considering how these structures can be used in a computing sense we need one more piece of terminology:

- Each item in a tree is called a 'node'

Thus the first item would be called the '*Root Node*' or '*Node 1*'.

Binary trees

It may now have occurred to you that one way of describing a 'list' is as 'a tree structure in which each parent is allowed only one descendant'. The simplest type of tree structure, other than a list, is one in which we restrict the maximum number of descendants a particular node may have, to two.

Such a structure is called a 'Binary Tree' and it turns out to be a very useful structure indeed. One application of Binary Trees is in the sorting and searching of large amounts of data. Since these applications probably occupy more computer time than any other single application it is little wonder that a vast amount of work has gone into ways of creating very efficient routines. The complexity of these published standard solutions, developed and refined over many years, tends to cloud the basics and make it extremely difficult for newcomers to come to terms with the more fundamental aspects involved.

By starting with fairly humble beginnings we can, hopefully, avoid these 'apparent complexities' and enable you to see the benefits that exist in using such structures .

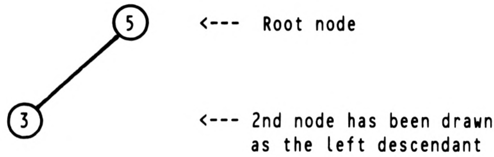
Consider the list of numbers '5, 3, 1, 6, 4'. We are going to place them onto a tree structure according to the following rules:

- If the number being added is *less than or equal to the value of the node being examined* then we shall move to the left descendant.
- If the number being added is 'greater than the value of the node being examined' then we shall move to the right descendant.
- When no suitable descendant exists then the number being added to the tree will be added in that vacant descendant position.
- We shall take the list of numbers in left to right order and this means that the first number (5) becomes the root node.

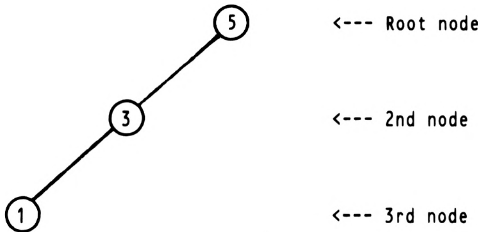
If we go through an example step by step you will get the general idea:

```
(5)   <----- This is the root node
             ( i.e.  1st node )
```

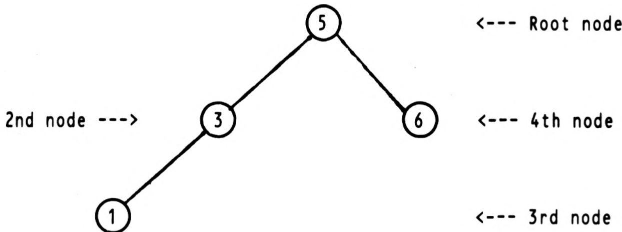
The next number in the list is 3 . Since our tree has only one item on it and therefore has no descendants we ask; 'is 3 less than or equal to 5 ?' Since it is, we shall draw 3 as the left descendant of the root node as follows:



The third item in the list is the number 1. To place this in its correct position, according to our rules, we proceed as follows: We compare the new entry (number 1), with the root node. Since 1 is less than the *value* of the root node we examine the left descendant of the root node (node number 2), the second item that we added to the tree. We ask 'is the value of the new entry less than or equal to the value of node number 2?'. Since it *is*, we see if node 2 has a left descendant. It hasn't and so this is where our new entry, the number 1, will be stored:

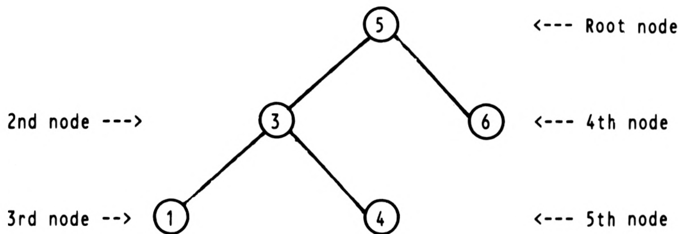


The fourth item in our list is the number 6. We do exactly the same as before and compare this value with the value of the root node. In this case the value is greater than that of the root node value. Since there is not a right descendant of our root node at present we proceed by making our fourth new entry the right descendant as follows.



Make quite sure you are clear about the terminology because it often causes problems. The numbering of the nodes themselves is dependent on the order that we are placing the items onto the tree. When we compare values,in order to ascertain where particular items should be placed, we are interested in the actual *value* that a particular node will have.

Let us place the last item in our list onto our tree. The item is the number 4. We compare the number 4 to the value of the first node in our tree. Since 4 is less than 5 we move to the left descendant of the first node. This is node number 2 which has a value of 3. We ask 'is 4 less than or equal to 3?'; it is not, so seeing that there is not a right descendant of this node we complete our tree by making the last entry the right descendant of node number 2.

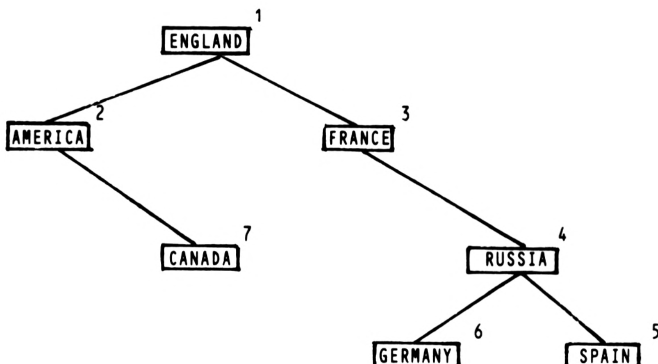


You have now created a 'sort tree' and at a first glance you may well be wondering what use such a structure can be. It can be noticed that the leftmost item on the tree is in fact the one with the lowest value. It is also apparent that the rightmost item is in fact the one with the largest value. Other than that there does not appear to be anything special about the arrangement.

Before we continue, try and draw a sort tree for the following list. This time we will consider a list of seven words:

ENGLAND
AMERICA
FRANCE
RUSSIA
SPAIN
GERMANY
CANADA

Use exactly the same rules as we did before, and apply them to the *alphabetic* rather than numeric order. You should end up with the following tree structure:



It is convenient in general to write the value of a node in a circle or rectangle and then in the top right above it put the node number as has been done in the previous example (SPAIN, for example, is node 5).

If you are still unsure about how to draw a sort tree from a list of numbers or words then write a few of your own lists and draw out their corresponding sort trees. Do it until you are quite clear in your own mind about the processes involved.

Two points should be noted in passing: firstly it was purely an arbitrary decision to make the 'less than or equal to' decision correspond to the 'left descendants' in the tree. We could equally have used the reverse convention. Secondly we could have split the decision part into 'less than' and 'equal to or greater than'. Again it was purely arbitrary.

What, however, *is* important is that the *way* in which we split the decision part enabled us to classify all incoming items into only one of two types. Thus there is never any doubt about the exact position that an incoming item will occupy on an existing tree.

Let us look at two other ways in which we could represent such a tree structure. Firstly we could represent it as a 'Table'. Look at figure 10.4 , it shows in table form the tree structure that we obtained with our list of numbers:

NODE	DATA ITEM	PARENT	LEFT	RIGHT
1	5	0	2	4
2	3	1	3	5
3	1	2	0	0
4	6	1	0	0
5	4	2	0	0

Figure 10.4: **Table representation of tree data**

Such a table is easily handled in many high level languages. In BASIC, for instance, it would be possible to define an array using statements like DIM T(5,4) if you wished to store the above table. In this instance T(n,1) would refer to the nth node's value; T(n,2) the node that is the parent of the nth node; T(n,3) would be the left descendant of the nth node, and so on. Later on we will look at a BASIC program (and some Z-80 assembly language routines) that will produce these type of tables.

Another way of representing a tree structure is with a different type of table as shown in figure 10.5. Supposing that for a tree containing N items we labelled N columns as the Nodes 1, 2, 3 .... N; and labelled N rows similarly as the Nodes 1, 2, 3 .... N. We could specify that a node, 'p' say, was the Parent of Node 'q', by placing a 1 in the table position (p,q). This type of representation is

normally called a *bit map* or a *relation matrix*. One advantage of such a description is that it can, by various techniques, be made very compact. Another advantage is that we can use matrix algebra to manipulate the relations. One disadvantage is that to make use of this type of representation you need some quite complex programming, not because of problems with the concept of a tree itself, just because it is a characteristic of this particular way of representing them.

PARENTS \ DESCENDANTS	1	2	3	4	5
1	0	1	0	1	0
2	0	0	1	0	1
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0

Figure 10.5: **Bit map representation of figure 10.4**

The Relation Matrix representation is not a good way to develop an *understanding* of the basic concepts of tree structures and so we shall not discuss them further.

For most applications of Binary Trees it is possible to use a table representation similar to that described earlier. Let us now consider a BASIC program using the table-based tree structure, an understanding of a high level language form will help us understand what must be done to create an assembly language equivalent.

A BASIC 'TREE TABLE' PROGRAM

Now that you have worked through the creation of some simple binary trees and have drawn their corresponding table forms, you will appreciate something of the approach:

- We start out by finding if there are any items on the tree at all.
- If there are not then all we have to do is make sure that the item being added becomes the root node.
- If there *are* items already, then the incoming item must be compared in exactly the same way as we did when drawing it as a picture.

The following BASIC examples use *Microsoft* BASIC – a very widely used version of the language. We are going to create a routine called 'Create tree' which can later be used as a general utility.

We will use a variable called `N%` to hold a record of the *number of items* that will be present in the finished tree. The variable `NEW.NODE%` is used to represent the *number of the item we are adding*. The *data items* themselves wil be placed in a vector variable called `DATA.ITEM$()`.

Line numbers are fairly arbitrary.

To place the first item on our tree we simply copy the item into `DATA.ITEM$(1)`. The element is specified by the node number held in a further variable `NEW.NODE%`. The first level of the 'Create table' subroutine will look like this:

```
490 rem ============================================================
500 NEW.NODE% = 1: GOSUB 5000       'Collect the first input item
550 FOR NEW.NODE% = 2 TO N%
560    GOSUB 5000                   'Collect next input item
580    GOSUB 1000                   'Store new item on tree
590 NEXT NEW.NODE%
600 RETURN
610 REM ============================================================
```

We must not forget that we need to define memory space for the table. To do this we shall, for the present, assume that the finished program will include a `DIM`ension statement like this:

```
10 DIM DATA.ITEM$(N%), PARENT%(N%)
15 DIM LEFT.DESCENDANT%(N%), RIGHT.DESCENDANT%(N%)
```

The following code is not written in the most concise way possible, but is laid out to enable you to relate the various sections to the earlier drawings we made of tree structures – the same reason accounts for the long and descriptive variable names.

The building of our tree really starts with subroutine 1000. As yet this is undefined but we do know what it will have to do: it will compare the incoming value with various nodes already in the tree. Since we will always have at least one new node in the tree before using subroutine 1000, we will not need to check whether the tree exists or not.

```
990 REM ===========================================================
1000 J%=1            ' We start at the 'Root' node i.e. node 1
1010 WHILE J% <> 0
1020      IF DATA.ITEM$(NEW.NODE%) <= DATA.ITEM$(J%)
THEN GOSUB 1200  ELSE GOSUB 1400
1030 WEND
1040 RETURN
1050 REM ==========================================================
```

Bear in mind that BASIC `WHILE .. WEND` loops and other condition test statements have an implied condition as part of the standard logic operations in BASIC. When we are testing '`WHILE J% <> 0`' we do not need to explicitly state the '`not equal to 0`' part but can instead simply write `WHILE J%`. For clarity we will write these conditions *ex*plicitly while looking at the various sections of code although the final example uses *im*plicit tests to save space.

We are using a variable `J%` to identify the position in the tree that we are examining. We compare the current item that is being placed on the tree with node `J%`. If the *value* of the new data item is *less than or equal to* the *value* of the node being examined then we must move down to the left descendant of the node being examined. If the *value* of the new data item is greater than the *value* of the node being examined then we will move down to the right descendant of the node being examined. These two alternatives are handled by two separate subroutines as follows:

```
1170 REM ============================================================
1180 REM  NEW ITEM IS LESS THAN OR EQUAL TO VALUE OF NODE J%
1190 REM ------------------------------------------------------------
1200 IF LEFT.DESCENDANT%(J%)<>0 THEN J%=LEFT.DESCENDANT%(J%):RETURN
1210 PARENT%(NEW.NODE%)=J%
1220 LEFT.DESCENDANT%(J%) = NEW.NODE%
1230 J%=0 'this forces us to leave WHILE / WEND loop in sub 1000
1240 RETURN
1250 REM ============================================================
....
....
1370 REM ============================================================
1380 REM  NEW ITEM IS GREATER THAN THAN VALUE OF NODE J%
1390 REM ------------------------------------------------------------
1400 IF RIGHT.DESCENDANT%(J%)<>0 THEN J%=RIGHT.DESCENDANT%(J%):RETURN
1410 PARENT%(NEW.NODE%) = J%
1420 RIGHT.DESCENDANT%(J%) = NEW.NODE%
1430 J%=0 'this forces us to leave WHILE / WEND loop in sub 1000
1440 RETURN
1450 REM ============================================================
```

Only one or the other of these subroutines will be called during the positioning of a given new item. Let us analyse the one corresponding to 'Less than or Equal to'. We have already tested the new data item value and the node `J%` and have found the value of the new item to be less than or equal to the value of the current node that is being examined (`J%`). Line 1200 looks to see whether node `J%` has got a left descendant. If it has, then we set `J%` to the value of `LEFT.DESCENDANT%(J%)` and then return from the subroutine 1200 via the `RETURN` statement on the same line.

The result of such an action is to return us into the `WHILE .. WEND` loop of subroutine 1000 where again we compare the data item that we are trying to place on the tree with what is now a new node `J%`.

We then repeat the process again and we are, in effect, moving down through the tree in just the same way as we did manually with our pictures.

If the test in line 1200 fails, ie. if there is no left descendant then we know straightaway that our new item is going to become that descendant. We also know that the current node J% is therefore going to be the parent of the new node that we shall create. In cases where the condition test in line 1200 fails we alter the left descendant pointer of node J% (which was zero) to the value NEW.NODE%. We also make node J% the parent of the new node. Since by this time the new item has been placed on the tree we set J% to zero. By doing this we will automatically leave the WHILE .. WEND loop which will still be in operation at the subroutine 1000 level. This will ensure that we fall through this level back to subroutine 500 ready to 'pick up' the next item to be placed onto the tree.

Subroutine 1400 operates in a similar fashion but concerns itself with those occasions that involve moving to a right descendant.

If we are to retrieve data in ascending order from our tree we must be able to locate the *lowest* value node that is present. From some of the tree structures you have already drawn, you have probably guessed that all you need to do is to start at the root node and keep going left until you run out of descendants. We write a subroutine called 'Lowest-node' to do this with a single line of BASIC as follows:

```
1420 REM ============================================================
2990 REM       L O W E S T - N O D E - S U B R O U T I N E
2995 REM ------------------------------------------------------------
3000 WHILE LEFT.DESCENDANT%(J%)<>0:J%=LEFT.DESCENDANT%(J%):WEND
     :RETURN
3030 REM ============================================================
```

We enter the above subroutine with J% as the number of the root node. The lowest value node is returned in J% overwriting the original value. By writing the subroutine in this way we can use it to find the lowest node of a special section of a tree known as a '*subtree*'. Consider any tree and then consider a particular node n as being the root of a smaller tree – that tree is the *subtree of node* n. We also talk about *left* and *right* subtrees. These are the subtrees formed by considering, respectively, the left and right descendants of a node as root nodes.

Before we can construct a program using these routines we must consider one last problem. Given a particular node we want to be able to find the node that is next in ascending order and print it. To do this we *do not* have to consider any of the values of the data items themselves because the order can be deduced from the parent, left descendant and right descendant pointers provided by our 'Create table' subroutine.

If you consider some drawn examples you will convince yourself that any subtree formed using the right descendant of a particular node n will only contain values greater than the value of node n. If we search this subtree for the node of lowest value we will have found the item that is next in order. Remember that we have already developed a subroutine to search a tree for the lowest value and we will be able to use this to search our sub-trees as well.

It is always possible that the current node being examined will not have a right descendant. If this is the case we must move up to the parent of the current node and repeat the process, but if we are moving up from a right descendant we must ignore this parent and move up again because the node will already have been printed. If during this 'climbing back' we find ourself at the root node then we will have printed all the nodes in the tree.

It is very helpful to relate these ideas to diagrams of various trees. Draw various subtrees in different colours and work through the above ideas on paper. As you develop a mental picture of how we are selecting the next item to print you will find the coding easier to follow.

The essential details of 'Next-node' are as follows: we look at the current node and ask 'is there a right descendant'? The coding is done like this:

```
3090 REM ============================================================
3100 IF RIGHT.DESCENDANT%(J%)<>0 THEN GOSUB 3200 ELSE GOSUB 3300
3110 RETURN
3120 REM ============================================================
```

Corresponding to the two possibilities we choose between two subroutines. If there is a right descendant then we move to it and then use 'Lowest-node' to find the lowest valued node of this right subtree:

```
3190 REM ============================================================
3200 J%=RIGHT.DESCENDANT%(J%):GOSUB 3000' Lowest-node
3220 RETURN
3230 REM ============================================================
```

If a right descendant does not exist we have to move up the tree. We keep track of the previous value of J% so that we can check whether we have moved upwards from a right descendant:

```
3290 REM ============================================================
3300 IF J%=1 THEN EXIT.FLAG%=0:RETURN
3310 OLD.J%=J%:J%=PARENT%(J%):
     IF RIGHT.DESCENDANT%(J%)=OLD.J% THEN GOTO 3300
3320 RETURN
3330 REM ============================================================
```

The completed subroutine can be seen in the listing of the example program where the above sections of code have been combined into a single block entitled 'Next node'.

The listing

We have developed three fairly simple subroutines that enable us to:

- Build a table corresponding to a tree structure
- Find the lowest node of a tree or subtree
- Find the next node in ascending order

These subroutines have been combined into a short program which:

- Builds a tree table using input from the terminal
- Prints the table, using a simple loop, so that you can examine it
- Prints the input data in ascending order using calls to 'Lowest node' and to 'Next-node'

The final code uses some multiple line statements to save space. Implied tests such as `WHILE J%`, rather than the explicit `WHILE J%<>0`, have also been used to shorten some of the lines of code. Figure 10.6 shows an example of the output that the program provides.

```
1 REM ==============================================================
2 REM                     S E T - U P - B L O C K
4 REM ______________________________________________________________
5 CLEAR:WIDTH LPRINT 70:INPUT"How many items do you wish to store";N%
10 DIM DATA.ITEM$(N%),PARENT%(N%),LEFT.DESCENDANT%(N%),
   RIGHT.DESCENDNT(N%)
25 REM =============================================================
30 GOSUB 500  '                                     Build tree table
32 REM =============================================================
33 REM                    P R I N T - T A B L E
34 REM _____________________________________________________________
35 LPRINT"Tree table has been created as ....":LPRINT
37 LPRINT "NODE","DATA ITEM","PARENT","LEFT","RIGHT"
40 FOR I% = 1 TO N%
50 LPRINT I%,DATA.ITEM&(I%);:
55 LPRINT TAB(30)
   PARENT%(I%).LEFT.DESCENDANT%(I%),RIGHT.DESCENDANT%(I%)
60 NEXT I%
```

```
80 REM ==============================================================
90 REM             P R I N T - D A T A - I N - O R D E R
100 REM -------------------------------------------------------------
105 LPRINT:LPRINT"Ordered list is as follows..........":LPRINT
110 J%=1:GOSUB 3000 '                                    Lowest-node
115 EXIT.FLAG%=1
120 WHILE EXIT.FLAG%
130 LPRINT DATA.ITEM$(J%),:GOSUB 3100 '                    Next-node
150 WEND
160 END ' ................. Logical end of program
460 REM =============================================================
470 REM         C R E A T E - T A B L E - S U B R O U T I N E
480 REM -------------------------------------------------------------
500 NEW.NODE%=1:GOSUB 5000 '                         Input data item
550  FOR NEW.NODE% = 2 TO N%
560    GOSUB 5000 '                                  Input data item
570    GOSUB 1000 '                     Second level of this routine
580  NEXT NEW.NODE%
590  RETURN
1000 J%=1
1010 WHILE J%
1020      IF DATA.ITEM$(NEW.NODE%) <= DATA.ITEM$(J%) THEN GOSUB 1200
          ELSE GOSUB 1400
1030 WEND
1040 RETURN
1200 IF LEFT.DESCENDANT%(J%) THEN J%=LEFT.DESCENDANT%(J%):RETURN
1210 PARENT%(NEW.NODE%)=J%:LEFT.DESCENDANT%(J%) = NEW.NODE%:J%=0
     :RETURN
1400 IF RIGHT.DESCENDANT%(J%) THEN J%=RIGHT.DESCENDANT%(J%):RETURN
1410 PARENT%(NEW.NODE%)=J%:RIGHT.DESCENDANT%(J%)=NEW.NODE%:J%=0
     :RETURN
1420 REM ============================================================
2990 REM       L O W E S T - N O D E - S U B R O U T I N E
2995 REM ------------------------------------------------------------
3000 WHILE LEFT.DESCENDANT%(J%)<>0:J%=LEFT.DESCENDANT%(J%):WEND
     :RETURN
3030 REM ============================================================
3080 REM        N E X T - N O D E - S U B R O U T I N E
3090 REM ------------------------------------------------------------
3100 IF RIGHT.DESCENDANT%(J%) THEN GOSUB 3200 ELSE GOSUB 3300
3110 RETURN
3200 J%=RIGHT.DESCENDANT%(J%):GOSUB 3000 '              Lowest-node
3220 RETURN
3300 IF J%=1 THEN EXIT.FLAG%=0:RETURN
3310 OLD.J%=J%:J%=PARENT%(J%):IF RIGHT.DESCENDANT%(J%)=OLD.J% THEN
     GOTO 3300
3320 RETURN
3330 REM ===========================================================
4980 REM             I N P U T - S U B R O U T I N E
4990 REM ------------------------------------------------------------
5000 INPUT "Enter value to be stored   ";DATA.ITEM$(NEW.NODE%):RETURN
5010 REM ============================================================
```

NODE	DATA ITEM	PARENT	LEFT	RIGHT
1	PAUL	0	2	3
2	ANDY	1	6	4
3	RUTH	1	12	28
4	GEORGE	2	9	5
5	MABEL	4	7	39
6	AMANDA	2	8	35
7	JENSINE	5	10	11
8	ALBERT	6	0	0
9	FRANK	4	15	14
10	IAN	7	31	21
11	JOSEPH	7	13	20
12	PETER	3	0	19
13	JOHN	11	38	0
14	FRED	9	0	0
15	CYRIL	9	16	18
16	CHRIS	15	22	17
17	CHRISTINE	16	0	0
18	DAVE	15	0	25
19	RONALD	12	33	29
20	KEVIN	11	26	0
21	JANICE	10	37	0
22	ANNE	16	23	24
23	ANN	22	0	0
24	CAROLINE	22	34	0
25	DAVID	18	0	0
26	JOYCE	20	0	27
27	JUDY	26	0	30
28	WENDY	3	40	0
29	RUSS	19	0	0
30	JULIE	27	0	32
31	HAROLD	10	0	0
32	KAY	30	0	0
33	ROBERT	19	0	36
34	BOB	24	0	0
35	ANDREAS	6	0	0
36	ROLF	33	0	0
37	JACK	21	0	0
38	JILL	13	0	0
39	MAUREEN	5	0	0
40	SANDRA	28	0	0

```
Ordered list is as follows..........

     ALBERT    AMANDA    ANDREAS   ANDY      ANN
     ANNE      BOB       CAROLINE  CHRIS     CHRISTINE
     CYRIL     DAVE      DAVID     FRANK     FRED
     GEORGE    HAROLD    IAN       JACK      JANICE
     JENSINE   JILL      JOHN      JOSEPH    JOYCE
     JUDY      JULIE     KAY       KEVIN     MABEL
     MAUREEN   PAUL      PETER     ROBERT    ROLF
     RONALD    RUSS      RUTH      SANDRA    WENDY
```

Figure 10.6: **Output from a BASIC tree sort program**

Some design notes

If you have not used 'tree sorts' before then the speed of these routines will be very impressive. Of particular interest is that we achieve the sorting without physically re-arranging any of the data items by using pointers to specify the logical structure of the corresponding tree.

The decision to use the 'parent' pointers needs a certain justification. Those of you who have come across these types of sorts before will be aware that it is standard practice to eliminate the parent pointers thus saving 33% of the total pointer space. It is also common practice to use recursion to provide some very elegant routines. These and other refinements such as relation matrix representation, when considered collectively, do a great job at hiding the essential simplicity of the basic concepts.

Those of you who have found the ideas straightforward you may like consider how the parent pointers can be eliminated. Look at the coding for the subroutines 'Lowest node' and 'Next node'. There is only one place that we actually use the parent pointers (program line 3310). If, within these two routines, we created a list of nodes that we encounter as we climb down a tree then we could climb back up by reading the list backwards. In this case we would only need to create left and right pointers in our table. Such a space saving becomes increasingly more important as the size of the data set that we are dealing with increases.

ASSEMBLY LANGUAGE TREE STRUCTURES

The key to writing assembly language tree sorts lies in understanding the principles behind the sort. We cannot in a book of this nature get too involved with the writing of comprehensive tree sort programs since much depends on the data formats used and on details of the application concerned. What we *can* do is provide you with plenty of clues that you can relate to our earlier discussions so that you acquire a 'feel' for the problems involved.

We set the scene for our discussion by considering the following problem. A block of text is present in memory (perhaps a memory image of a text file read into memory from diskette or tape). The text is to be read sequentially and each time an 'end of word' is detected a descriptor is to be built giving the character count, the start address of the word and the addresses of the descriptors of the left and right descendants. Parent pointers are *not* going to be used. The definition of 'end of word' is to some extent arbitrary but a reasonable choice is that the end of a word is found when a space or any other non alphabetical character is detected.

Descriptor format

Figure 10.7 shows a suitable format for the descriptor. Seven bytes per word are therefore required to create the text tree data structure.

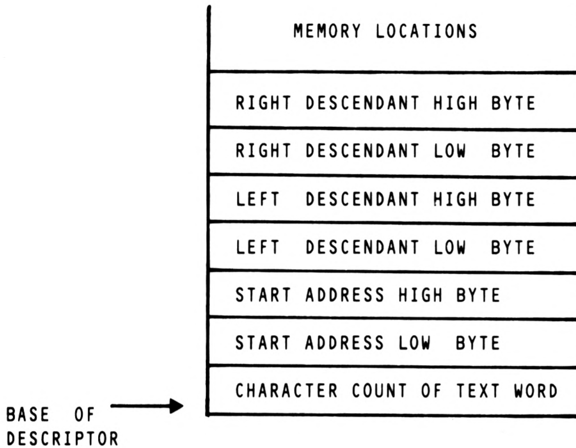


***Figure 10.7:* Layout of a text tree descriptor**

A block of descriptors can be built starting from the top of the text so that the the tree structure is created without physically re-arranging any of the text at all.

The start of the block of descriptors is the start of the tree, the very first descriptor created is the 'root node'. To add a second item is easy: the word being *added* is compared to the root node word. If the new word is alphabetically *less than or equal to* the root then the second entry becomes the left descendant of the root. To do this all that needs to be done is to place the start address of the second descriptor into the *left descendant* area in the first descriptor. Conversely if the new word is *not* alphabetically less than or equal to the root word then we make the second entry the *right descendant* of the root node.

Let us assume that a tree exists of at least one node. In a similar fashion to that shown in chapter 9 we can create a DE–C set for the new word to be added to the tree. If index register IX holds the start of a descriptor then an equivalent HL–B set for a node word can be created using indexed addressing as follows:

```
LD   B,(IX+0)        ;Count value
LD   L,(IX+1)        ;Low byte of start address
LD   H,(IX+2)        ;High byte of start address
```

To place a new item on the tree we want to start at the root node, compare the new word with the current *node word* (root to start with). If the new word is less than or equal to the node word we move to the left, if greater we move to the right.

The following loop illustrates the general ideas involved:

```
CREATE$NODE: LD   IX,TEXT$ROOT$NODE    ;Base of first tree descriptor
C$N$1:       LD   B,(IX+0)             ;Count value
             LD   L,(IX+1)             ;Low byte of start address
             LD   H,(IX+2)             ;High byte of start address
             CALL COMPARE$WORDS        ;Carry set if HL-B is greater
             CALL C,LEFT$DESCENDANT    ;C set - node word > new word
             CALL NC,RIGHT$DESCENDANT
             JR   NZ,C$N$1             ;Zero flag is exit condition
```

To create a descendant we place the address of the new descriptor being created into the descriptor that is the current *node being examined*. If in performing the calls to LEFT$DESCENDANT and RIGHT$DESCENDANT we find that no further descendants exist then the current node being looked at is going to become the parent of the new node being added. We have implied in the above code that if this condition is found the zero flag will be set to force an exit from the loop.

Routines for left and right descendants are fairly similar so let us just look at the left version. We use a PUSH AF instruction to preserve the status of the carry flag, then we look at bytes (IX+3) and (IX+4) to see if a descendant exists. If a descendant does exist we set IX to the address of the start of the left descendants descriptor, otherwise we create a new left descendant. The following code assumes that HL already contains the start address of the new descriptor being created:

```
* ==================================================================
LEFT$DESCENDANT:    PUSH AF          ;Must preserve carry
                    LD   A,(IX+3)    ;Low byte of left descendant
                    OR   (IX+4)      ;Z flag set if no descendant
                    JNZ  MOVE$TO$LEFT
* ------------------------------------------------------------------
CREATE$LEFT:        LD   (IX+3),L
                    LD   (IX+4),H
                    POP  AF
                    RET
* ------------------------------------------------------------------
MOVE$TO$LEFT:       LD   E,(IX+3)    ;Get low byte
                    LD   D,(IX+4)    ;Get high byte
                    PUSH DE          ;Transfer trick to
                    POP  IX          ;perform IX <--- DE
                    POP  AF          ;Restore carry status
                    RLA              ;Carry into bit 0 of accumulator
                    CP   A           ;Trick to clear zero flag
                    RR               ;Bit 0 back into carry
                    RET
* ==================================================================
```

This is a *double ended* subroutine that includes two tricks. Firstly the PUSH DE, POP IX instructions are used to provide a means of transferring the contents of DE into index register IX. Secondly RLA, CP A , RR is used to protect the carry flag whilst setting the zero flag.

Routines such as these are used to build the descriptor block that defines the logical structure imposed on the text image in memory. In order to use such a structure it is necessary to write subroutines that perform functions similar to those BASIC subroutines 'Lowest-node' and 'Next-node' that were developed during our tree sort introduction.

Without getting too involved we will develop one routine to show you the principles. The assembly language equivalent of 'Lowest-node' has to search the tree for the left-most item. We start at the root (the first descriptor) and look to see if (IX+3) and (IX+4) are zero. If they are then no further left descendants exist and therefore thus the current node is the lowest node on the tree. If a descendant does exist then we set IX to the address of the left descendants descriptor and repeat the loop until such time as we run out of left descendants. A simple routine providing the essential ideas is as follows:

```
* ==================================================================
LOWEST$NODE:    PUSH AF ! PUSH DE  ;Preserve registers
L$N$1:          LD   A,(IX+3)      ;Low byte of left descendant
                OR   (IX+4)        ;Zero flag set if no L. descendant
                JR   NZ,L$N$2      ;Have we found it ?
                LD   E,(IX+3)      ;Low byte of left descendant
                LD   D,(IX+3)      ;High byte of left descendant
                PUSH DE            ;Copy trick
                POP  IX
                JR   L$N$1         ;Keep going left
L$N$2:          POP  DE ! POP AF   ;Restore registers
                RET
* ==================================================================
```

If parent pointers were being used this routine would be satisfactory. Without parent pointers this routine would correctly find the *lowest* node, but other routines such as 'Next node' would not be able to operate because there is no way of moving *up* the tree. The solution is to track all movement through the tree using a *stack* (a LIFO structure). Any movement through the tree is monitored by pushing the address of the current descriptor (the current node) onto the tree-stack. Many approaches are possible, here is an easy one that uses the Z-80's own stack mechanism. We save the existing Z-80 stack pointer, then load the SP register with our own *tree stack pointer* held in a location TREE$SP say. We then use the Z-80 PUSH IX instruction to place our current descriptor address onto the tree stack. Finally we save the new value of the tree stack pointer and re-instate the Z-80 stack. A typical modification is shown below:

```
* ================================================================
*                L O W E S T - N O D E
* ================================================================
LOWEST$NODE:   PUSH AF ! PUSH DE    ;Preserve registers
L$N$1:         LD   A,(IX+3)        ;Low byte of left descendant
               OR   (IX+4)          ;Zero flag set if no L. descendant
               JR   NZ,L$N$2        ;Have we found it ?
               CALL ADD$TO$STACK    ;Track movement through tree
               LD   E,(IX+3)        ;Low byte of left descendant
               LD   D,(IX+3)        ;High byte of left descendant
               PUSH DE              ;Copy trick
               POP  IX
               JR   L$N$1           ;Keep going left
L$N$2:         POP  DE ! POP AF     ;Restore registers
               RET
* ----------------------------------------------------------------
ADD$TO$STACK:  LD   (Z80$SP),SP     ;Save Z80 stack
               LD   SP,(TREE$SP)    ;Get tree pointer
               PUSH IX              ;Store this node on tree stack
               LD   (TREE$SP),SP    ;Save tree pointer
               LD   SP,(Z80$SP)     ;Restore Z80 stack
               RET
* ================================================================
```

Routines wishing to climb the tree can therefore do so by popping the necessary addresses from the tree stack. The general principle is that you PUSH descriptor addresses onto the stack whenever you move down the tree, and POP descriptor addresses whenever you move up the tree. The stack approach is one good example of a structure that is in fact easier to implement in assembly langauage than it is in languages such as BASIC.

Hopefully we have given you some insight into tree structures. The ideas are sometimes difficult to grasp on initial contact but rest assured that they do become easier once the essential characteristics of such structures are understood. Hopefully you now appreciate what a tree structure is, and you should be aware of the types of problems that need to examined when using them.

11 SOLVING PROBLEMS

Certain things make the development of assembly language programs less problematic than they can be, we examine some of these areas in this chapter.

DOCUMENTATION

We have dealt in detail with various design aspects and this goes hand in hand with another frequently neglected item – documentation. By this we do not just mean details of how the program is to be used, we also mean development notes, details of amendments, program notes and so on. Programmers in general are noted more for their 'Let's do some coding' attitudes than for any excessive desire to document their programs – eventually failure to keep adequate notes will cost dearly, both in lessons not learned and in lost time.

The golden rule is simple:

- Document *while* you are developing the program, *not* afterwards

By all means tidy up the development notes after the program is complete but don't wait this long before you make any notes at all.

If possible try to develop a more or less standard layout for all your projects. You need development notes which will show, when taken in conjunction with any design work, what the objectives of writing the program were, and which explain the reasons behind your approach. You need sufficient program details to enable you *and others* others to understand the operation of the program. Finally, if the program is to be used by non-technical people you will also need 'jargon free' user instructions.

The task of producing this documentation is not as difficult as it might seem. If you have a text editor program then you can keep most of the documentation on tape or diskette, this has the advantage of being very easy to keep up to date. In many cases you will be able to use the editor program that comes with your assembler to prepare your documentation. Here are some guidelines:

Keep all your design diagrams and make notes about the problems you encounter during the development. Make special note of any assumptions made which might affect program operation if changed in the future. Note also which parts of the code are dependent on things like the operating system I/O characteristics and even particular control characters which might vary from system to system.

With short routines *include the documentation with the source code*. If the programs are larger, then use your editor program to create a separate documentation file.

Keep essential details within the source code itself to tell you where the additional documentation may be found, when the program was written and so on. A simple scheme is usually all that is required and a typical source code 'header' is shown in figure 11.1.

```
* ===================================================================
* Program Name........... S P E L L - C H E C K
* ===================================================================
*
* Project Reference.....81/A11/1-1
* Copyright (C) 1981 by Paul Andreas Overaa
*
*  Purpose....... First phase of a  CP/M based spelling check program
*                 using  simple memory  based  tree sort  techniques.
*                 Will  operate on  all  common   ASCII based  files,
*                 including Wordstar type.
*
*  Date.......... Project start 23rd January  1981
*
*                 This source   12th February 1981
*
*  Processor..... Either Z80 or 8080/8085
*
*  Operating system.... CP/M 1.4 onwards
*
*  Documentation notes.... Kept separate from source,
*                          see project ref file.
*
*  Diskette  forms... Program and documentation are both available:
*                    (in RAIR S/D format)
*
*                          Source code        SPELL.ASM
*
*                          Object code        SPELL.COM
*
*                          Documentation      SPELL.DOC
*
* ===================================================================
```

Figure 11.1: **A typical program header**

If a routine requires a particular format for the data that it works on then provide some sort of indication within the routine itself so that the general ideas behind it are apparent. Use a title that indicates what operation the routine performs – Figure 11.2 shows a typical example.

```
* ================================================================
*                 W O R D - S E A R C H   (Revision A)
* ----------------------------------------------------------------
* Purpose:
* This routine searches the text file for words. Each time the start
* of a word is found a 'count' of  the number of  characters begins.
* When the  end of  the  word   (any non-alphabetical character)  is
* identified Word-search calls a routine to create a new node  which
* contains the starting address of the word, its character count and
* the descriptor addresses of the left and right nodes in the binary
* tree that is created. All 'nodes' created have an associated block
* of SEVEN bytes  called the  node descriptor. If a  word  is  found
* that is already present on the tree the  node creation is aborted.
* The resulting descriptor set is therefore free of any duplication.
* ----------------------------------------------------------------
WORD$SEARCH:            LD      BC,0            ;Zero B and C together
                        LD      DE,0
                        LD      HL,(CURRENT$DMA$ADDRESS)
                        LD      (NEXT$EMPTY$TEXT$NODE),HL
                        LD      (TEXT$ROOT$NODE),HL
                        LD      HL,0
                        LD      (TEXT$NODE$COUNT),HL
                        LD      HL,PROGRAM$TOP  ;Start of file buffer
W$S$1:                  LD      A,(HL)          ;Get a character
                        AND     7FH             ;Clear bit 7
                        CALL    IDENTIFY$CHARACTER
                        CP      EOF
                        JR      NZ,W$S$1
                        RET
* ================================================================
IDENTIFY$CHARACTER:  CALL  UPPER$LOWER$CASE
                     CALL  C,LETTER$ADJUSTMENT     ;Must preserve HL
                     CALL  NC,NON$LETTER$ADJUSTMENT;and carry for
                     INC   HL                      ;CC part
                     RET
* ================================================================
```

Figure 11.2: **Try to include brief explanation at the start of a routine**

STANDARD PROGRAM LAYOUT

In the same way that a standardized documentation layout helps to provide consistency, so does a standardized program layout. Even if all your programs are different there are many things about the overall structure which will often be similar.

All will have some type of 'initial block', where EQUates are defined, stacks initialized and so on. Most will use various system calls which may need your own routines to 'wait for input', to pass data to the operating system in a particular way and to perform other functions. Additionally it may be necessary to reserve areas within the program for particular uses such as static data areas.

Figure 11.3 shows a typical assembly language source code layout. Bear in mind that this representation means that the assembled programs will have the data areas at the top of the object code as figure 11.4 illustrates.

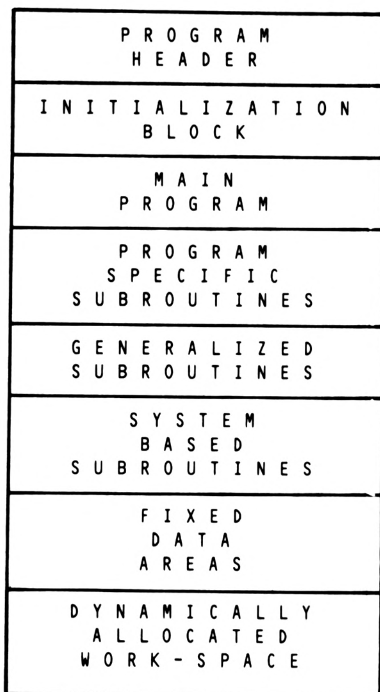


Figure 11.3: **Suggested plan for a source code layout**

LIBRARY ROUTINES

Many routines find use over and over again in a wide range of different programs. The advantages of building up a subroutine library is two-fold:

- First it is not necessary to re-write the routines, they can usually be loaded from tape or cassette directly into the source code you are writing

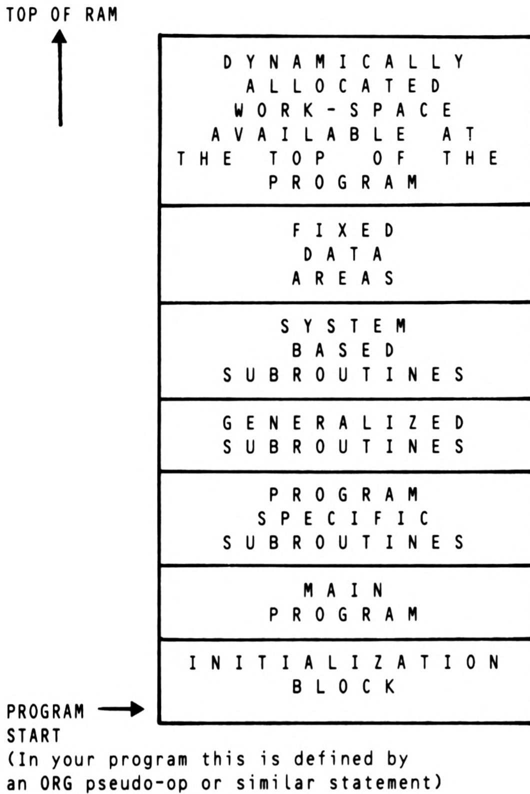


***Figure 11.4:* Resulting layout of object code in memory**

- Second, you will feel more comfortable using such routines because you know they have been tried and tested

It goes without saying that you should not include routines in your 'library' until you are happy that they actually do the job that they are supposed to. In practice the availability of such pre-written routines will greatly increase the speed at which your programs become operational.

COMMON PROBLEMS

The golden rule is *don't panic*, and do not start making undocumented changes to your source code either. Errors are, on the whole, easily approachable; they come in various shapes and sizes, often under the general heading of *syntax errors*.

Syntax errors are, or should be, by far the most common problem. You may mis-spell the name of a label (such as `CHECK$WRD` when

you meant CHECK$WORD), or mis-type an instruction mnemonic (LD,A B). You may write a comment without the delimiter character which tells your assembler to ignore it. Doing this is a great way to upset your assembler, because it will probably try to assemble your comment as though it was a series of instructions – obviously it does not get very far before it finds 'instructions' that it cannot understand!.

These errors are usually picked up by the assembler when you try to assemble the source code. The messages that you will get vary but the type of thing to expect is the 'UNDEFINED LABEL', 'BAD OP CODE', 'DUPLICATE LABEL' and the like. The exact forms will be found from your assembler manual and are usually self explanatory. Often an assembler will create a separate file showing the errors that have been found and their locations. Facilities vary substantially from assembler to assembler.

Other common errors can include relative jumps that are greater than allowed and the incorrect use of various assembler directives. Your assembler manuals should provide full details of the types of error that will be identified.

PROBLEMS AFTER ASSEMBLY

Just because your source code has been assembled without errors it does not necessarily mean that the program will work. It only means that the assembler has been able to understand the instructions and has translated them into the appropriate object code (or sometimes into an intermediate 'hex' code). If for example you have written a jump on zero instruction mnemonic (JP Z) when you really meant to write jump on *not zero* (JP NZ), then your program will not function correctly. Again, prevention is the best policy: try to plan out and design small sections of code using subroutines for as many procedures as possible. Small sections of code are easier to write, easier to examine when things go wrong, and easier to change if the need arises.

The isolation of distinct subroutines has two benefits in terms of debugging:

- It is usually possible to devise ways of checking particular routines one at a time

- It is much easier to alter or replace a small subroutine than it is to untangle a small section of assembly code buried in a large mass

If a program or routine does not run as expected then try to identify the area in the *source code* which is likely to be causing the problems. Examine the code to ensure that the instructions written are the instructions that you intended to write. Are you testing the right

flag? Is the comparison test the wrong way around? Have you forgotten to include a RETurn instruction at the end of a subroutine?.

Try to identify the cause of the problem *before* you alter the source code. Above all do not be tempted to guess! If a routine does not work then there is a definite fault somewhere. You should (with practice) be able to identify that fault from the source code listing, and until then it is advisable not to make any wild guesses, they simply lead to further problems.

LOGICAL FAULTS

These problems are frequently the most disastrous. There is nothing worse than spending many hours writing routines which, once combined, are found to be riddled with seemingly inexplicable new faults. When faults such as this occur there is no simple answer, the debugging of these types of problems often requires large sections of code to be changed (often introducing further errors!).

The best way to avoid the problems that logical faults create is to use techniques that enable the design of the program to be viewed during development. This is why we place so much emphasis on the Warnier diagram as a design tool.

OVERALL PLAN

- Identify the problem and keep a written description of your project
- Break the problem down, using the techniques we have described, so that smaller and more manageable areas can be identified
- Make notes on how you expect to code those areas that appear straightforward
- Ask yourself whether more complex areas can be broken down still further. The answer is usually 'Yes'. Continue this breakdown process until you are happy about the *coding* of each section of the project
- Use the design diagrams and your notes as a starting point for a basic program structure. Make lists of the routines that need to be developed, and make a note of those which may be available from your library
- Work on individual routines in isolation, but keep in mind that they must be compatible with the remainder of the program

- Test routines individually to ensure that they perform as they should. Complex programs can be built up piece by piece with any errors usually resulting from the most recently added routine

12
LINKING INTO BASIC

In this chapter we move away from pure assembly language programs and give an example of how small assembler routines can be 'hidden' in some rather unusual places. These tricks are sometimes used for efficiency, sometimes for speed, and often for more dubious reasons.

It is possible to create your own assembly language routines for use in a program written primarily in a high level language. If you are using a compiler then sophisticated link techniques are usually available. If you use interpreted BASIC then other methods are necessary. The most popular example is probably the POKEing of routines into BASIC's REM statements. Since REM's are not used in any way a routine can safely reside within the characters of the line, without causing any problems.

You can also load assembly language routines into variables, provided that you take certain precautions. The overall ideas behind all of these related techniques are similar and we shall look at two examples using Microsoft BASIC. Firstly we show you how to hide an assembly language subroutine within a string variable, then we will show how you can incorporate a 'memory mapped screen save' routine into your own BASIC programs.

Microsoft BASIC includes the CALL statement which allows you to call an assembly language subroutine. In its simplest form the CALL statement is followed by the starting address of the subroutine. However, indiscriminate use of CALL will cause the system to 'crash', so it is wise to exercise a certain amount of care when using it. With operating systems such as CP/M you can actually damage diskette data if you inadvertently execute one of the disk operating system routines – indeed it is not unusual for some people to use these effects as part of their 'program protection' techniques.

Before using the CALL statement we must understand a little about the state of the microprocessor at the time a call is executed. When a call is made you can safely assume that all of the internal registers contain information which must be preserved. It is therefore necessary to push the contents of all of these registers on to the stack. If, as is usual, you are only using the primary set of Z-80 registers then you only need to save these.

Usually a BASIC CALL statement will preserve the contents of the necessary registers automatically, but there may be some BASICS available that expect you to do it from within your own routines. We shall therefore assume, at least to start with, that Microsoft BASIC does not preserve the register contents when a CALL statement is encountered. Having developed some initial routines we will then be able to check whether we really need to save the internal registers, or whether it is being done for us.

The versions of Microsoft BASIC that we have worked with allow for only 16 bytes of stack storage in a user-called subroutine. If more stack space is needed then BASIC's stack pointer must be stored, and the stack pointer initialized so that an independent stack is available to the called routine. The contents of the stack pointer must be restored to their original value before returning control to the BASIC interpreter. A typical routine, if we do not need to set up additional stack space, will take the form illustrated in figure 12.1, and it is this scheme that will be used in our example program.

```
        PUSH AF            ; save internal registers
        PUSH BC
        PUSH DE
        PUSH HL

           ↑

  BODY OF SUBROUTINE

           ↓

        POP HL             ; restore  original contents
        POP DE
        POP BC
        POP AF
        RET                ; and RETurn to BASIC
```

Figure 12.1: **General format required for a simple CALL**

To force a routine into a variable (or into a REM statement) we must translate the mnemonic form into a numeric form. This is done by using the equivalent op-codes, addresses and data values. Such exercises make useful educational material, and certainly help us appreciate the benefits of modern assemblers!

On the *Osborne*-01 Z-80-based computer the VDU screen is cleared by 'printing' the character $1A_{hex}$. The Osborne uses CP/M as its operating system and to output a character it is necessary to do three things:

- The character to be 'output' must be loaded into the E register
- The number 2 must be loaded into the C register (this is called the 'function number' and signifies to CP/M that character output is required)
- A CALL instruction must be issued using a common entry point at location 05_{hex}. This entry point is usually labelled BDOS by CP/M programmers (the label stands for Basic Disk Operating System).

If we add the three instructions to the basic format shown in figure 12.1, and list each instruction together with the hexadecimal equivalent we obtain the form shown in figure 12.2.

Mnemonic form	*Hexadecimal form*
PUSH AF	F5
PUSH BC	C5
PUSH DE	D5
PUSH HL	E5
LD E,1AH	1E, 1A
LD C,2	0E, 02
CALL BDOS	CD, 05, 00
POP HL	E1
POP DE	D1
POP BC	C1
POP AF	F1
RET	C9

***Figure 12.2:* Translation of program into hexadecimal form**

The routine, if it were to be assembled, would be translated by the assembler to the binary equivalent of the set of numbers F5, C5, D5, E5, 1E, 1A, 0E, 02, CD, 05, 00, E1, D1, C1, F1, C9. We will place these numbers directly into a portion of memory which we will assign to a 'dummy' string variable.

Microsoft's VARPTR() function is used to obtain addresses of variables to enable them to be passed to assembly language routines etc. If we make an assignment X$=STRING$(50,0) then we define a *string variable* of 50 characters, each of which has the ASCII code 0 – we will have a string of 50 null characters. The function VARPTR(X$) does not return directly the address we need, instead it returns the address corresponding to the first byte of a three byte 'pointer'.

The format of this pointer is needed if we are to find the *actual* address of the variable. Here is how it is arranged:

- The first byte is the number of characters in the string
- The second byte is the low order part of the address
- The third byte is the high order part of the address

If the variable 'I' is used to collect the result of the VARPTR() function then we can define a variable ADDRESS like this:

```
ADDRESS=PEEK(I+1)+PEEK(I+2)*256
```

We multiply the high byte of the address by 256 to left shift it eight bits higher than the low order part. If this seems confusing try writing out some examples and translating them by hand. Remember that addresses on the Z-80 are stored low byte first, so that PEEK(I+1) is reading the low order byte.

The first program we show uses the hexadecimal numbers as a DATA statement. We read the values and poke them into space reserved for the X$ variable whose starting address has been found as indicated above. Having done this our routine is present inside X$ and can be used by an appropriate CALL statement. Here is a simple trial program:

```
10 REM ===========================================================
20 REM  USING MICROSOFT 'STRING SPACE' FOR AN ASSEMBLY ROUTINE
30 REM -----------------------------------------------------------
40 X$= STRING$(60,0)            ' Dummy variable will hold the routine
50 DATA &HF5,&HC5,&HD5,&HE5:REM ......hex forms of PUSH instructions
55 DATA &H1E,&H1A:REM ................hex form of LD E,1AH
56 DATA &H0E,&H02:REM ................hex form of LD C,2
57 DATA &HCD,&H05,&H00:REM ...........hex form of CALL 0005H
60 DATA &HE1,&HD1,&HC1,&HF1,&HC9:REM .hex forms of POP and RETurn
70 I=VARPTR(X$)     ' First byte is the character count for the string
80 ADDRESS=PEEK(I+1)+PEEK(I+2)*256            ' See text about this line
90 FOR J=ADDRESS TO ADDRESS+15
100 READ X:POKE J,X          ' Read data value and place into position
110 NEXT J
120 CALL ADDRESS             ' This uses routine that we placed into X$
130 END
140 REM ===========================================================
```

The instructions in the DATA statements have been separated for clarity. There is, however, no reason why the hex numbers cannot be written as a single DATA declaration if desired.

AN EXAMPLE: DUPLICATING A DISPLAY FILE

For some applications it is necessary to reserve space explicitly outside BASIC itself and facilities usually exist to set the highest

address which will be used by BASIC. This may done by placing the necessary value into a specific location (the ZX-81 has a system variable called `RAMTOP` which may be used in this way). Other BASICS, like Microsoft's, have initialisation options which will reserve external space. When Microsoft BASIC is loaded from disk, the command `MBASIC /M:&HBB00` will cause BASIC to load into memory, once in use it will not use *any* memory space above the specified $\texttt{BB00}_{hex}$.

We shall make use of the ability to reserve RAM space to allocate an area for copying the contents of the display file of a memory mapped system. The particular example taken again involves the *Osborne*-01, but the general principles will be similar with any memory mapped system, only the actual addresses used will change.

Figure 12.3 shows the schematic memory layout of the *Osborne*-01 when using Microsoft BASIC. The important point is to notice that the display file occupies addresses $\texttt{F000}_{hex}$ to $\texttt{FFFF}_{hex}$ – it is situated at the *top* of the memory.

Memory areas between $\texttt{CB00}_{hex}$ and $\texttt{F000}_{hex}$ are reserved for the operating system and obviously must not be used.

When BASIC loads it does so from $\texttt{100}_{hex}$ upwards and will use locations from $\texttt{CB00}_{hex}$ downwards for storage of variables, strings etc. Since we can restrict the area of memory used by Microsoft BASIC we will load using `MBASIC /M:&HBB00` to reserve 4K of memory in which we will duplicate the contents of the screen (our display file).

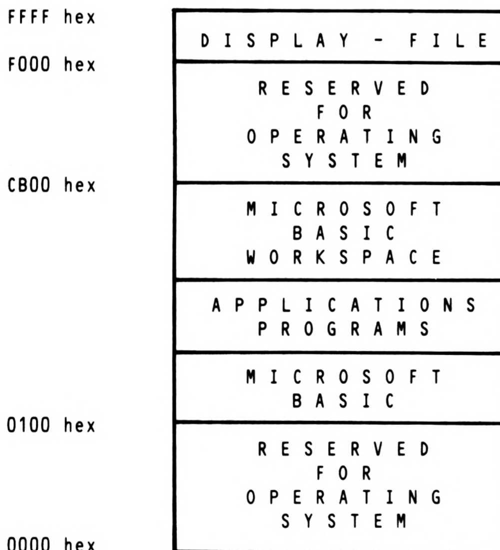


***Figure 12.3:* Schematic layout of the *Osborne*-01 with BASIC**

The equivalent memory organization obtained when reserving space in this way can be seen from figure 12.4. Space reserved can of course provide a 'common area' for assembly language subroutines, data transfers between different applications programs and has many other uses.

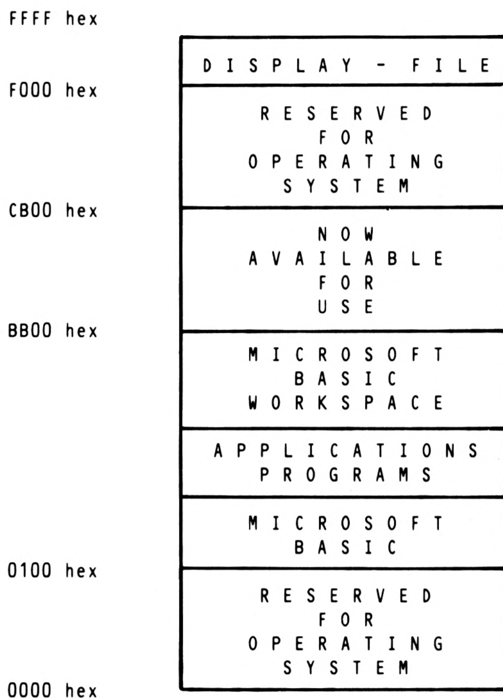


***Figure 12.4:* Memory layout with space for screen copy**

The copy routine itself is very straightforward and uses one of the powerful block move instructions LDIR. The BC register pair is loaded with the number of bytes to be transfered, this is $FFFF_{hex}$ *minus* $F000_{hex}$, which is FFF_{hex}. The HL pair is loaded with the starting address of the source block – $F000_{hex}$ in our example. Finally the DE pair loaded with the starting address of the destination block – $BB00_{hex}$. The instruction LDIR will automatically copy the source block into the destination; the contents of the byte addressed by HL are transferred to the byte addressed by the DE register pair, then DE and HL are incremented and BC is decremented. If BC is not equal to zero then the program counter is decreased by 2 and the instruction repeated. The main part of the routine simply requires three 'load' instructions and the block move instruction. As with the first example we shall store the routine in a

string variable, the data to be transferred will be passed outside BASIC to the area $BB00_{hex}$ to $CB00_{hex}$ which has been reserved for the screen copy. As with the first example we list the mnemonic forms of the instructions (including the stack save/restore operations needed for the CALL operation) and convert these into their hexadecimal form so that the routine can be POKEd into position.

Figure 12.5 shows the operations required and the equivalent hexadecimal form to be used in the DATA statement. When writing routines such as these, do not be tempted to skip the 'mnemonic stage'. Op codes are easily forgotten or transposed, so write the program just as if you were going to assemble it, *then* translate it into numeric form. When such routines are incorporated into BASIC programs you should ensure that you document the program sufficiently. You may understand a routine when you write it, but what about next week, next month, or next year?

Mnemonic form	*Hexadecimal form*
PUSH AF	F5
PUSH BC	C5
PUSH DE	D5
PUSH HL	E5
LD BC,0FFFH	01, FF, 0F
LD HL.,F000H	21, 00, F0
LD DE BB00H	11, 00, BB
LDIR	ED, B0
POP HL	E1
POPDE	D1
POP BC	C1
POP AF	F1
RET	C9

***Figure 12.5:* Translation of screen copy program to hexadecimal form**

A simple test program which places this routine into a string variable (X$) is shown below. Note the similarity in layout to the earlier program. Remember that the routine which *performs* the copy is *inside* the variable X$, the contents of the screen are placed into the area of memory reserved when we first loaded BASIC.

```
10 REM ==============================================================
20 REM    S C R E E N  -  C O P Y  -  R O U T I N E
30 REM ______________________________________________________________
40 X$=STRING$(20,0)                 ' Routine is placed into this space
50 DATA &HF5,&HC5,&HD5,&HE5:REM .......hex forms of PUSH instructions
60 DATA &H01,&HFF,&H0F:REM ...........hex form of LD BC 0FFFH
```

```
70 DATA &H21,&H00,&HF0:REM ...........hex form of LD HL F000
80 DATA &H11,&H00,&HBB:REM ...........hex form of LD DE BB00H
85 DATA &HED,&HB0:REM ................hex form of LDIR
90 DATA &HE1,&HD1,&HC1,&HF1,&HC9:REM ..hex forms of POP and RETurn
100 I=VARPTR(X$)   '  First byte is the character count for the string
110 ADDRESS=PEEK(I+1)+PEEK(I+2)*256         ' See text about this line
120 FOR J=ADDRESS TO ADDRESS+19
130 READ X:POKE J,X           ' Read data value and place into position
140 NEXT J
150 CALL ADDRESS              ' this call saves current screen contents
160 END
170 REM ===========================================================
```

The example illustrates the general principles and should provide some food for thought for the more devious readers. Routines within variables can be saved and read to disk or tape just like any other variable.

Preserving the registers – Is it necessary?

If you are lucky you may find this information in the language manuals, but more often than not you will have to find out for yourself. The easiest way is simple – try it and see: write a short routine which changes values of the microprocessor registers, then use it with and without the PUSH and POP instructions. The program below has had the PUSH and POP instructions removed. Remember that the loop counter (currently allowing for 11 iterations) must be changed to reflect the smaller number of bytes now being POKEd into the string variable.

```
10 REM ============================================================
20 REM    S C R E E N  -  C O P Y  -  R O U T I N E
30 REM ------------------------------------------------------------
40 X$=STRING$(20,0)' Routine is placed into this space
60 DATA &H01,&HFF,&H0F:REM ...........hex form of LD BC 0FFFH
70 DATA &H21,&H00,&HF0:REM ...........hex form of LD HL F000
80 DATA &H11,&H00,&HBB:REM ...........hex form of LD DE BB00H
85 DATA &HED,&HB0:REM ................hex form of LDIR
90 DATA &HC9:REM .....................hex form of RETurn
100 I=VARPTR(X$)'First byte is the character count for the string
110 ADDRESS=PEEK(I+1)+PEEK(I+2)*256' See text about this line
120 FOR J=ADDRESS TO ADDRESS+11
130 READ X:POKE J,X' Read data value and place into position
140 NEXT J
150 CALL ADDRESS' this call saves current screen contents
160 END
170 REM ===========================================================
```

The results of this test indicated that Microsoft disk-based BASIC *does* preserve and restore the necessary registers; it is quite likely that your version of BASIC will do the same but, if there is any doubt, it is far better to check for yourself.

FINAL WORD

There is much scope for this type of 'mixed code' programming within normal applications packages. Often you can use assembler to do things which require maximum speed such as sorts and graphic displays, but you can still benefit in other areas by having the high level language available. The level of such compromise can be adjusted to suit your own time and complexity requirements. If you are feeling adventurous here are some possible approaches you might like to develop:

- Write a routine that stores the 'calling program' to disk or tape. Some commercial sorting programs use this technique so that the maximum amount of RAM, less the operating system and the sort routine itself, is made available for the actual sort.

- Write a routine to convert upper case to lower case and lower case to upper case.

- Using external reserved RAM space create a 'help menu' that is entered by *not* responding to a request for input within a certain time. Use an assembler routine or BASIC functions such as INKEY$ to identify the time taken to respond to input.

NOTES

13
WHERE NEXT?

In this last chapter we want to briefly consider some of the implications of the ideas contained within this book. We start by covering some points about Warnier diagrams that have been, to a certain extent, avoided.

VERIFYING YOUR DESIGN

By now you have seen some of the uses that Warnier diagrams may be put to and we have tried to illustrate some of the ways in which such diagrams may be used to describe the structure of data and of programs themselves. The emphasis has been centred around the separation of the logical problems of *programming* from the physical problems of actually *coding* the solution for a particular language or a particular computer. We have dealt primarily with the Z-80 microprocessor, but many of the fundamental ideas are not processor dependent.

Many of you will have wondered, whilst reading this book, what happens if you make a logical error as you prepare a Warnier diagram. Such errors will sometimes occur but you will be less likely to make such mistakes because the diagrams represent your logical solution in a very 'pictorial' fashion. Frequently you will know that a fault exists just by looking at the diagram. You can then take steps to make the necessary modifications. Used in this way the Warnier diagram becomes a prop to lean on as you are working towards a solution.

It is possible to be more rigorous in the use of the concepts that we have looked at and since there exists a relation between the defined objectives of a problem, the correct Warnier representation of the problem, and the efficiency of the final implementation, it is relevant to consider one way to make sure your Warnier diagram is faultless.

In the early chapters we illustrated the use of the Warnier diagram as an iconic model for the logic we are attempting to describe or create. In doing this the Warnier diagram is actually mapping out the program structure required to implement our solution.

Warnier does not concern himself with these aspects because the use of these diagrams as an iterative design tool for analysing problems is not fundamental to his approach. Those of you who have studied any

of Warnier's works will realise that to a large extent he attains a correct logical solution using various techniques including Boolean Algebra, Karnaugh Maps and Decision Tables. Such solutions are then represented by a Warnier diagram. The program is then constructed from the diagram as indicated.

For our purposes the Warnier diagram is being used in a rather different way to that originally employed by Warnier himself. We use them to analyse and document our thoughts on a problem, that is to say the Warnier diagram is being used as a design tool to provide an iconic model helping us to achieve solutions by a process of *iterative refinement*.

It is sometimes helpful, when using the Warnier diagrams in this way, to be able to verify the efficiency and correctness of your implied solution. One way to do this is to translate the diagram into an algebraic expression using the Algebra of Sets, Boolean Algebra or any other isomorphic algebra that you might be familiar with. To give you some insight into how this can be done we will take a simple example and describe what is done at each stage.

AN EXAMPLE

Let us take a very general example of a Warnier diagram and use the letters A, B and C to represent three conditional tests that are present in the structure of the program. Let us also define U1, U2, U3 and U4 as subsets of actions that are performed in accordance with the logical description shown in figure 13.1.

There is nothing special about the example other than the fact that it was made purposely inefficient. You can regard U1, U2 etc. as being subroutines which are called as desired. If for instance condition A is true and condition B is also true then the top third from left bracket will be performed. If in the course of carrying out the operations in this bracket the test C fails (ie. is *not true*) then subroutine U2 would be called. If the test C did not fail (ie. condition C was *true*), then subroutine U1 would be called instead.

We get a clue about verifying such a diagram from one of the ways that Warnier uses to solve his logic problems. At times he will get a solution from a decision table of possible options in terms of a Boolean Algebra expression. He then proceeds to describe the solution with a Warnier diagram. The implication is straightforward:

- If you can convert a Boolean expression into a Warnier diagram then you can convert a Warnier diagram back into a Boolean expression.

Having done that, you can manipulate the expression and reduce it

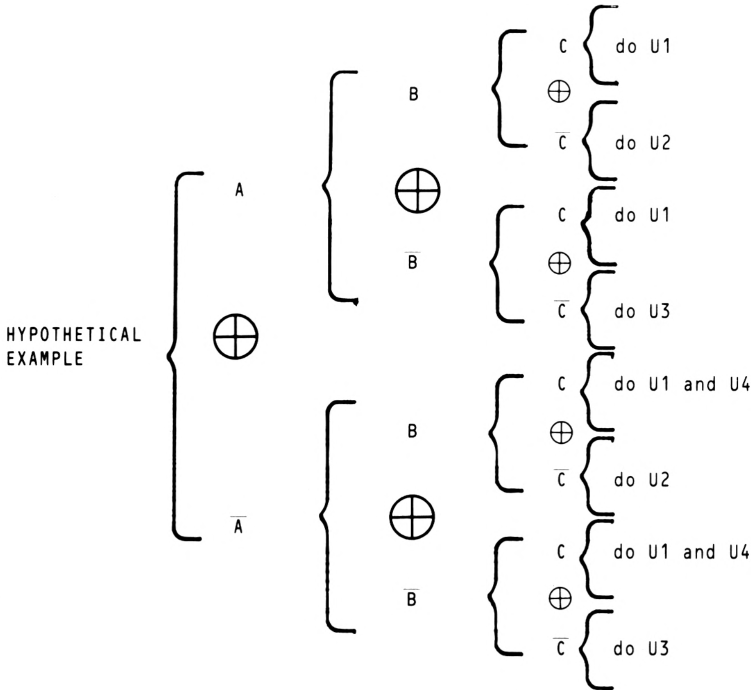


Figure 13.1: **Warnier diagram for a hypothetical example**

to its simplest form (or confirm that it is already in its simplest form). It is then perfectly easy to take the simplified expression and convert it back to the Warnier diagram form. The resulting diagram will then be correct and will represent the simplified logical solution.

If you study figure 13.1 you will see that subroutine U2 is called in two places:

- If test A is true and test B is true but test C is not true then U2 will be called.
- If test A is not true and test B is true and test C is not true then again subroutine U2 will be called.

We can easily express the fact U2 is dependent on these two condition requirements in the following way:

$$U2 = A.B.\overline{C} + \overline{A}.B.\overline{C}$$

This is a Boolean algebra expression of the set of conditions under which subroutine U2 is called. We can, in a similar fashion, write

down expressions for all of the subroutines U1 to U4. If we do this we get the following results:

$U1 = A.B.C + A.\overline{B}.C + \overline{A}.B.C + \overline{A}.\overline{B}.C$

$U2 = A.B.\overline{C} + \overline{A}.B.\overline{C}$

$U3 = A.\overline{B}.\overline{C} + \overline{A}.\overline{B}.\overline{C}$

$U4 = \overline{A}.B.C + \overline{A}.\overline{B}.C$

The notation is derived from Boolean Algebra but the way you describe the expressions in words is up to you. U2 can be described as the subroutine that is carried out when either 'A and B are true but C is not true', or 'B is true but A and C are not true'.

To follow the reduction of the above expressions all you need to be aware of is the fact that you can treat the letters on the right hand side just as you would treat unknowns in an equation. The object of the exercise is to regroup the symbols so that we can bracket together complementary terms such as A and $\overline{A}$ because we can then eliminate them.

Look first at U2 and follow through the reduction:

$U2 = A.B.\overline{C} + \overline{A}.B.\overline{C}$

Note that $B.\overline{C}$ is common to both expressions and re-arrange accordingly:

$U2 = B.\overline{C}\ (A + \overline{A})$

This immediately leads to the reduced expression for U2 as:

$U2 = B.\overline{C}$

Now we try to reduce U3 in a similar way:

$U3 = A.\overline{B}.\overline{C} + \overline{A}.\overline{B}.\overline{C}$

$U3 = \overline{B}.\overline{C}\ (A + \overline{A})$

$U3 = \overline{B}.\overline{C}$

With U4 we proceed as follows:

$U4 = \overline{A}.B.C + \overline{A}.\overline{B}.C$

$U4 = \overline{A}.C\ (B + \overline{B})$

$U4 = \overline{A}.C$

Lastly we can reduce U1 in the following manner:

$U1 = A.B.C + A.\overline{B}.C + \overline{A}.B.C + \overline{A}.\overline{B}.C$

$U1 = A.C\ (B + \overline{B}) + \overline{A}.C\ (B + \overline{B})$

$U1 = A.C + \overline{A}.C$

$U1 = C\ (A + \overline{A})$

$U1 = C$

We have now simplified all of the original expressions and have obtained the following results:

$U1 = C$

$U2 = B.C$

$U3 = \overline{B}.\overline{C}$

$U4 = \overline{A}.C$

How do we convert these expressions back into an efficient Warnier diagram. The first thing to do is to re-arrange the expressions so that the most frequent condition test comes first on the right hand side, then comes the next most frequent – and so on. If we do this we obtain the following forms:

$U1 = C$

$U4 = C.\overline{A}$

$U2 = \overline{C}.B$

$U3 = \overline{C}.\overline{B}$

Look closely at the way the reduced forms have been arranged and then look at the Warnier diagram in figure 13.2 We can draw the diagram directly from the re-arranged Boolean expressions.

You will notice that we have effected quite an improvement on the logical structure of our hypothetical program. If we consider some of our earlier thoughts we can see some useful concepts emerging. We can use Warnier diagrams to pictorially represent our problem as we come to terms with the various constraints and can create a 'picture' of our logical solution. We can also check the validity of a solution by translating the diagram into algebraic form and attempting to reduce the expressions we obtained. If we find reduction *is* possible then by translating back we can improve the original solution. The final Warnier diagram will describe the necessary structure of the program in a way that is easy to translate into computer code.

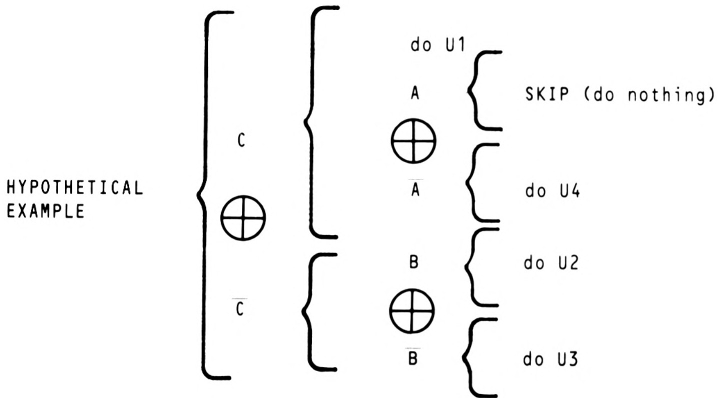


Figure 13.2: **Improved diagram for the example**

The correspondence between a Warnier diagram and Set Algebra or the isomorphic Boolean Algebra provides a link into the realms of mathematics with many implications concerning the correctness of the structure of a program.

SYSTEM PLANNING

It is of interest to make one last connection concerning the uses of the Warnier diagram. As you know we can regard a program as a set of instructions. We can divide such a set into subsets and represent the inherent structure using a Warnier diagram. It is equally advantageous in systems design to consider the system as being divisible into subsets of actions. Such a subset defines a set of logically related actions which may be combined into a program module.

Imagine, for instance, that we are designing a system around some routine business application. In practice we would need to be able to add and delete data, analyse it, print reports and so on. We might decide on a menu driven system and could describe the highest level menu in Warnier form as in figure 13.3 (here all options are mutually exclusive):

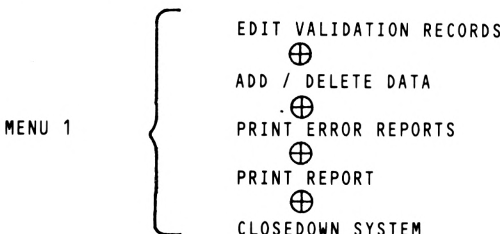


Figure 13.3: **Essential menu details**

Such a diagram indicates the bare essentials of what we want our system to do. Each term can obviously be expanded into much greater detail. The simple statement in figure 13.3 'EDIT VALIDATION RECORDS' can be expanded to incorporate some additional ideas as in figure 13.4.

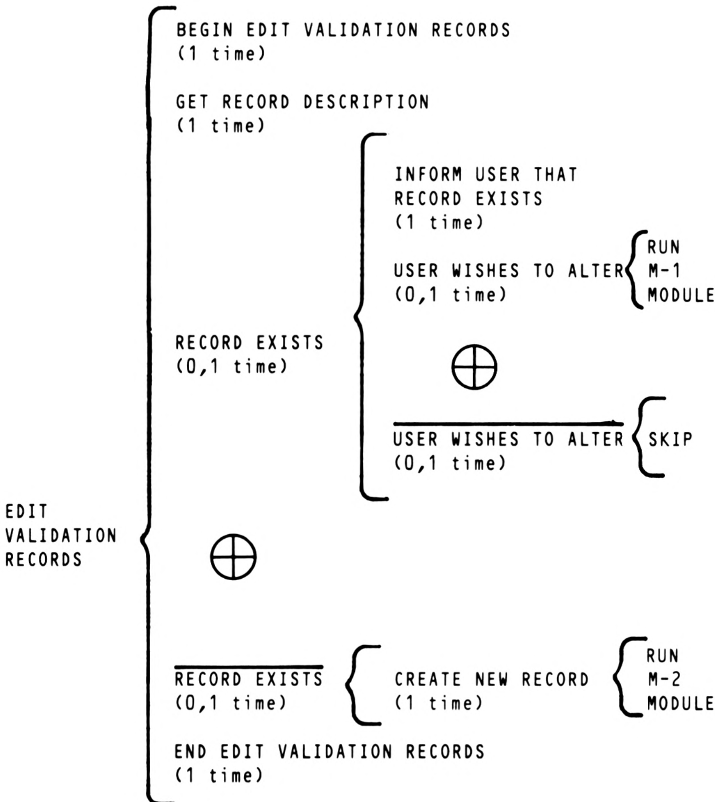


Figure 13.4: **Edit validation records, further expansion**

We can, by using these ideas, see that there is no fundamental difference between designing a system and designing a program. It is just as easy to develop a logical coherent system as it is to develop a logical program.

We have tried throughout the book to emphasise some of the new ideas which seem to be of use in our quest for better methods of writing and designing programs. There is no doubt that the work of Jean Dominique Warnier is of fundamental importance in this search. We dealt initially with some ideas connected with how we solve problems and the usefulness of having 'pictures' or 'iconic

models' to relate to. We have also seen how the basic concepts of a set can provide interesting and useful descriptions that are of particular use when programming in assembly language.

This last chapter has dealt briefly with one approach to verifying your solutions and suggests that the design of systems or sets of programs is really no different from designing programs themselves.

COMPACT CODING

Throughout the book you will notice that we have not emphasised the saving of bytes. Neither have we suggested that the efficiency of a program should be judged by its compactness alone.

It is nevertheless true to say that it is important not to waste memory unneccessarily. Indeed some applications may require the use of many legitimate tricks to squeeze as much useful code as possible into a given space. If it is necessary to put a 5K operating system into a 4K ROM you must cut corners, you will therefore forced to save bytes by using many tricks that will inevitably make your code convoluted and extremely difficult to understand. Such solutions are governed by the necessity of cramming as much code as possible into a given area. Where our ideas may possibly differ is that we would be inclined to do such compression *after* we had the overall design identified and not before or during the development.

LAST WORD

By avoiding some of the difficult areas, such as hardware interfacing and the problems of microprocessor I/O, we have hopefully provided you with a fairly gentle introduction to assembly language programming. Our final piece of advice is simply this:

- As you examine more advanced texts bear in mind that behind each program or routine there is, or should be, a sound logical design.
- Try to understand the basis of the routines you examine and try also to separate and understand the logical structure as something quite separate from the coding itself.
- Remember that writing obscure code for its own sake is rarely wise. At the very least you may need to understand it many years after it was written, equally likely is the possibility that someone else may need to!

Today, good programming does not just involve efficient code, it involves producing efficient code in a reasonable amount of time and it involves producing code that can be easily altered (and maintained) should the need arise. The techniques that we have illustrated are identical whether they are being used for system planning, high level programming, or low level programming. Obviously such ideas do not solve all problems and many other useful techniques and approaches exist.

If you are new to computing then use the ideas that you have understood and concentrate on the underlying essentials. If you are not a beginner then you can be assured that the concepts covered may be taken much further. We hope that the ideas have provided food for thought. Perhaps, like us, you will consider that the unity of some of the underlying concepts may indicate that it is no longer necessary to regard good programming as magic or as an art-form. Good programming can be taught just as easily as we teach other subjects – providing we use the right techniques and ensure that the underlying fundamental ideas are understood.

NOTES

Appendix A
THE Z-80 INSTRUCTION SET

The following instruction set listing is based on the data available from the manufacturer's data sheets and from other sources. The notation used is based on generally accepted Z-80 mnemonic conventions as follows:

Symbol	**Description**
r, rl	Any 8-bit register
rr	A register pair eg. BC, DE
s	An 8-bit operand. Note that this may be a register, an immediate byte of data or an address from which the operand is actually obtained. Note also that such an address may be specified in several ways.
n	An 8-bit value
mn	A 16-bit value
d	An 8-bit value which is a 'displacement' in an indexed addressed instruction
(xx)	Brackets are used to indicate that it is the *contents* of byte xx that are being treated as the operand, and *not* the value xx itself. As an example the notation (HL) should be read 'the contents of the byte whose address is held in HL'.
Flags	0 = flag set to this value 1 = flag set to this value * = flag value depends on the result of the instruction ? = this is explained in notes below the flag description
pq	Any 16-bit address
$\overline{X}$	X bar is the complement of X, ie. all 1's are turned into 0's and all 0's turned to 1's *eg.* if X = 10110000 then $\overline{X}$ = 01001111
b	A specified bit within a byte or set of bytes
e	An 8-bit number in two's complement form
cc	An allowed condition to be tested *eg.* Z = zero NZ = not zero

Symbolic description of an instruction

It is possible to describe the effect of an instruction in notation form and this has certain advantages as shown in these two examples:

- The instruction LD r, n is shown in symbol form as r←n. This implies that the value n is loaded (copied) into register r.
- The instruction LD A, (HL) is shown in symbol form as A←(HL). This implies that the *contents* of the byte whose address is HL are loaded (copied) into the A register (the accumulator).

Where it is felt that a symbolic notation is not specifically of value (as in, for example, the CALL instructions), then only the text description has been given. The overall format of the instruction set in this appendix is as follows:

Mnemonic: This is the standard 'shorthand' description of the instruction. It is the form that your assembler will expect

Title: Mnemonic in 'word form'

Description: Brief details of the function of the instruction

Flags: Details of which flags are affected by the instruction

This information is followed by details of the allowed forms for the instruction. This includes the addressing modes, the assembly language forms for a given mnemonic, the object code resulting from a particular instruction and the number of bytes occupied by the assembled instruction.

Mnemonic: **ADC A, s**

Title: Add accumulator with carry

Symbolic: A←A + s + carry

Description: The operand and the carry flag are added to the value already present in the accumulator

Flags:

S	Z	H	P/V	N	C
*	*	*	*	0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	ADC A, A	8F	1
	ADC A, B	88	1
	ADC A, C	89	1
	ADC A, D	8A	1
	ADC A, E	8B	1
	ADC A, H	8C	1
	ADC A, L	8D	1
Immediate	ADC A, n	CE n	2
Indirect	ADC A, (HL)	8E	1
Indexed	ADC A, (IX + d)	DD 8E d	3
	ADC A, (IY + d)	FD 8E d	3

Mnemonic:	**ADC HL, rr**
Title:	Add with carry to HL
Symbolic:	HL←HL + rr + carry
Description:	The contents of the specified register pair together with the contents of the carry flag are added to the value already present in HL

Flags:

S	Z	H	P/V	N	C
*	*	*	*	0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	ADC HL, BC	ED 4A	2
	ADC HL, DE	ED 5A	2
	ADC HL, HL	ED 6A	2
	ADC HL, SP	ED 7A	2

Mnemonic:	**ADD A, (HL)**
Title:	Add indirect addressed location to accumulator
Symbolic:	A←A + (HL)
Description:	The contents of the byte whose address is specified by HL are added to value already present in the accumulator

Flags:

S	Z	H	P/V	N	C
*	*	*	*	0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	ADD A, (HL)	86	1

Mnemonic:	**ADD A, (IX + d)**
Title:	Add indexed addressed location (IX + d) to accumulator
Symbolic:	A←A + (IX + d)
Description:	The contents of the byte specified by (IX + d) are added to the value already present in the accumulator

Flags:

S	Z	H	P/V	N	C
*	*	*	*	0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indexed	ADD A, (IX + d)	DD 86 d	3

Mnemonic: **ADD A, (IY + d)**

Title: Add indexed addressed location (IY + d) to accumulator

Symbolic: A←A + (IY + d)

Description: The contents of the byte specified by (IY + d) are added to the value present in the accumulator

Flags:

S	Z	H	P/V	N	C
*	*	*	*	0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indexed	ADD A, (IY + d)	FD 86 d	3

Mnemonic: **ADD A, n**

Title: Add immediate data byte to the accumulator

Symbolic: A←A + n

Description: The contents of the byte that follows the op code is added to the value already present in the accumulator

Flags:

S	Z	H	P/V	N	C
*	*	*	*	0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Immediate	ADD A, n	C6 n	2

Mnemonic: **ADD A, r**

Title: Add register r to accumulator

Symbolic: A←A + r

Description: The contents of register r are added to the value already present in the accumulator

Flags:

S	Z	H	P/V	N	C
*	*	*	*	0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	ADD A, A	87	1
	ADD A, B	80	1
	ADD A, C	81	1
	ADD A, D	82	1
	ADD A, E	83	1
	ADD A, H	84	1
	ADD A, L	85	1

Mnemonic: **ADD HL, rr**

Title: Add register pair to HL

Symbolic: HL←HL + rr

Description: The contents of the specified register pair are added to the value already present in the HL pair

Flags:

S	Z	H	P/V	N	C
		*		0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	ADD HL, BC	09	1
	ADD HL, DE	19	1
	ADD HL, HL	29	1
	ADD HL, SP	39	1

Mnemonic: **ADD IX, rr**

Title: Add register pair to IX

Symbolic: IX←IX + rr

Description: The contents of the register pair rr are added to the contents of index register IX

Flags:

S	Z	H	P/V	N	C
		*		0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	ADD IX, BC	DD 09	2
	ADD IX, DE	DD 19	2
	ADD IX, HL	DD 29	2
	ADD IX, SP	DD 39	2

Mnemonic: **ADD IY, rr**

Title: Add register pair to IY

Symbolic: IY←IY + rr

Description: The contents of the register pair rr are added to the contents of index register IY

Flags:

S	Z	H	P/V	N	C
		*		0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	ADD IY, BC	FD 09	2
	ADD IY, DE	FD 19	2
	ADD IY, HL	FD 29	2
	ADD IY, SP	FD 39	2

Mnemonic: **AND s**

Title: AND accumulator with specified operand

Symbolic: A←A∧s

Description: The accumulator and the operand are compared bit by bit according to the following rule table:

	Accumulator bit	0	1
Operand Bit	0	0	0
	1	0	1

Result in accumulator

Flags:

S	Z	H	P/V	N	C
*	*	1	*	0	0

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	AND A	A7	1
	AND B	A0	1
	AND C	A1	1
	AND D	A2	1
	AND E	A3	1
	AND H	A4	1
	AND L	A5	1
Immediate	AND n	E6 n	2
Indirect	AND (HL)	A6	1
Indexed	AND (IX + d)	DD A6 d	3
	AND (IY + d)	FD A6 d	3

Mnemonic: **BIT b, (HL)**

Title: Test bit b of the memory location addressed by HL

Description: The selected bit is tested and the zero flag set if bit is zero

Flags:

S	Z	H	P/V	N	C
*	*	1	*	0	

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	BIT 0, (HL)	CB 46	2
	BIT 1, (HL)	CB 4E	2
	BIT 2, (HL)	CB 56	2
	BIT 3, (HL)	CB 5E	2
	BIT 4, (HL)	CB 66	2
	BIT 5, (HL)	CB 6E	2
	BIT 6, (HL)	CB 76	2
	BIT 7, (HL)	CB 7E	2

Mnemonic: **BIT b, (IX + d)**

Title: Test bit b of indexed addressed location (IX + d)

Description: The selected bit is tested and the zero flag set if bit is zero

Flags:

S	Z	H	P/V	N	C
*	*	1	*	0	

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indexed	BIT 0, (IX + d)	DD CB d 46	4
	BIT 1, (IX + d)	DD CB d 4E	4
	BIT 2, (IX + d)	DD CB d 56	4
	BIT 3, (IX + d)	DD CB d 5E	4
	BIT 4, (IX + d)	DD CB d 66	4
	BIT 5, (IX + d)	DD CB d 6E	4
	BIT 6, (IX + d)	DD CB d 76	4
	BIT 7, (IX + d)	DD CB d 7E	4

Mnemonic: **BIT b, (IY + d)**

Title: Test bit b of indexed addressed location (IY + d)

Description: The selected bit is tested and the zero flag set if bit is zero

Flags:

S	Z	H	P/V	N	C
*	*	1	*	0	

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indexed	BIT 0, (IX + d)	FD CB d 46	4
	BIT 1, (IX + d)	FD CB d 4E	4
	BIT 2, (IX + d)	FD CB d 56	4
	BIT 3, (IX + d)	FD CB d 5E	4
	BIT 4, (IX + d)	FD CB d 66	4
	BIT 5, (IX + d)	FD CB d 6E	4
	BIT 6, (IX + d)	FD CB d 76	4
	BIT 7, (IX + d)	FD CB d 7E	4

Mnemonic: **BIT b, r**

Title: Test bit b of register r

Description: The selected bit of register r is tested and the zero flag set if bit is zero

Note: This implied addressing instruction is two bytes long. The first byte is CB_{hex}, the second byte depends on which register and which bit is being tested. The following table shows the second byte op code possibilities:

Flags:

S	Z	H	P/V	N	C
*	*	1	*	0	

	REGISTERS	A	B	C	D	E	H	L
	0	47	40	41	42	43	44	45
	1	4F	48	49	4A	4B	4C	4D
B	2	57	50	51	52	53	54	55
I	3	5F	58	59	5A	5B	5C	5D
T	4	67	60	61	62	63	64	65
S	5	6F	68	69	6A	6B	6C	6D
	6	77	70	71	72	73	74	75
	7	7F	78	79	7A	7B	7C	7D

Mnemonic: **CALL condition, pq**

Title: Call subroutine at address pq if condition is satisfied

Description: If condition is met the program counter contents are placed on the stack. The specified address is then loaded into the program counter

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Immediate	CALL NZ, pq	C4 q p	3
	CALL Z, pq	CC q p	3
	CALL NC, pq	D4 q p	3
	CALL C, pq	DC q p	3
	CALL PO, pq	E4 q p	3
	CALL PE, pq	EC q p	3
	CALL P, pq	F4 q p	3
	CALL M, pq	FC q p	3

Mnemonic: **CALL pq**

Title: Call subroutine at address pq

Description: Unconditional subroutine call. The program counter contents are placed on the stack and the specified address is then loaded into the program counter

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Immediate	CALL pq	CD q p	3

Mnemonic: **CCF**

Title: Complement carry flag

Symbolic: $C \leftarrow \overline{C}$

Description: The carry flag is complemented

Flags:

S	Z	H	P/V	N	C
		*		0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	CCF	3F	1

Mnemonic: **CP s**

Title: Compare operand to accumulator

Description: The operand is subtracted from the accumulator. The result is not stored but is used to condition the status of the flag register

Flags:

S	Z	H	P/V	N	C
*	*	*	*	1	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	CP A	BF	1
	CP B	B8	1
	CP C	B9	1
	CP D	BA	1
	CP E	BB	1
	CP H	BC	1
	CP L	BD	1
Immediate	CP n	FE n	2
Indirect	CP (HL)	BE	1
Indexed	CP (IX + d)	DD BE d	3
	CP (IY + d)	FD BE d	3

Mnemonic: **CPD**

Title: Compare with decrement

Symbolic: Flags conditioned by A – (HL) : HL←HL – 1
: BC←BC – 1

Description: The contents of the byte indirectly addressed by HL are subtracted from the accumulator. The result is not stored but does condition the status of the flag register. HL and BC are then decremented

Flags:

S	Z	H	P/V	N	C
*	?	*	?	1	

Z flag set if A=(HL)
P/V flag reset if BC=0 after execution, else set

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	CPD	ED A9	2

Mnemonic: **CPDR**

Title: Block compare with decrement

Symbolic: Flags conditioned by A – (HL) : HL←HL – 1
: BC←BC – 1
Repeated until BC=0 or A=(HL)

Description: As CPD but if BC is non-zero and A does not equal (HL) then the program counter is decreased by two and the instruction re-executed

Flags:

S	Z	H	P/V	N	C
*	?	*	?	1	

Z flag set if A=(HL)
P/V flag reset if BC= 0 after execution, else set

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	CPDR	ED B9	2

Mnemonic: **CPI**

Title: Compare with increment

Symbolic: Flags conditioned by A – (HL) : HL←HL + 1
: BC←BC – 1

Description: Byte addressed by HL is subtracted from the accumulator. Result is not stored but flag register is conditioned. HL is then incremented and BC decremented

Flags:

S	Z	H	P/V	N	C
*	?	*	?	*	

Z flag is set if A=(HL)
P/V flag is reset if BC=0 after execution, else set

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	CPI	ED A1	2

Mnemonic: **CPIR**

Title: Block compare with increment

Symbolic: Flags conditioned by A – (HL) : HL←HL + 1
: BC←BC – 1
Repeated until BC=0 or A=(HL)

Description: As per CPDR but HL is incremented instead

Flags:

S	Z	H	P/V	N	C
*	?	*	?	1	

Z flag set if A=(HL)
P/V flag reset if BC=0 after execution, else set

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	CPIR	ED B1	2

Mnemonic: **CPL**

Title: Complement accumulator

Symbolic: $A \leftarrow \overline{A}$

Description: The contents of the accumulator are complemented

Flags:

S	Z	H	P/V	N	C
		1		1	

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	CPL	2F	1

Mnemonic: **DAA**

Title: Decimal adjust accumulator

Description: *See manufacturer's data*

Mnemonic: **DEC s**

Title: Decrease operand s

Symbolic: s←s – 1

Description: The contents of the specified byte are decremented

Flags:

S	Z	H	P/V	N	C
*	*	*	*	1	

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	DEC A	3D	1
	DEC B	05	1
	DEC C	0D	1
	DEC D	15	1
	DEC E	1D	1
	DEC H	25	1
	DEC L	2D	1
Indirect	DEC (HL)	35	1
Indexed	DEC (IX + d)	DD 35 d	3
	DEC (IY + d)	FD 35 d	3

Mnemonic: **DEC rr**

Title: Decrease register pair rr

Symbolic: rr←rr – 1

Description: The contents of the register pair rr are decreased by one

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	DEC BC	0B	1
	DEC DE	1B	1
	DEC HL	2B	1
	DEC SP	3B	1

Mnemonic: **DEC IX**

Title: Decrement IX

Symbolic: IX←IX – 1

Description: The contents of the IX register are decreased by one

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	DEC IX	DD 2B	2

Mnemonic: **DEC IY**

Title: Decrement IY

Symbolic: IY←IY – 1

Description: The contents of the IY register are decreased by one

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	DEC IY	FD 2B	2

Mnemonic: **DI**

Title: Disable interrupts

Description: *See manufacturer's data*

Mnemonic: **DJNZ e**

Title: Decrement B and relative jump on *not* zero

Symbolic: B←B – 1
If B <> 0 then PC←PC + e

Description: B register is decremented. If result is *not* zero then a relative jump is performed

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Immediate	DJNZ e	10 e	2

Mnemonic: **EI**

Title: Enable interrupts

Description: *See manufacturer's data*

Mnemonic: **EX AF, AF′**

Title: Exchange accumulator and flags with alternate registers

Symbolic: AF↔AF′

Description: Current 'active' accumulator and flag registers are interchanged with the alternate set

Flags:

S	Z	H	P/V	N	C
*	*	*	*	*	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	EX AF, AF′	08	1

Mnemonic: **EX DE, HL**

Title: Exchange contents of the DE and HL registers

Symbolic: DE↔HL

Description: The contents are effectively swapped

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	EX DE, HL	EB	1

Mnemonic: **EX (SP), HL**

Title: Exchange HL with top of stack

Description: *See manufacturer's data*

Mnemonic: **EX (SP), IX**

Title: Exchange IX with top of stack

Description: *See manufacturer's data*

Mnemonic: **EX (SP), IY**

Title: Exchange IY with top of stack

Description: *See manufacturer's data*

Mnemonic: **EXX**

Title: Exchange alternate registers

Symbolic:
BC←BC'
DE←DE'
HL←HL'

Description: The contents of the BC, DE and HL pairs are interchanged with the alternate set BC', DE' and HL'

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	EXX	D9	1

Mnemonic: **HALT**

Title: Halt CPU

Description: *See manufacturer's data*

Mnemonic: **IM 0**

Title: Set interrupt mode 0 condition

Description: *See manufacturer's data*

Mnemonic: **IM 1**

Title: Set interrupt mode 1 condition

Description: *See manufacturer's data*

Mnemonic: **IM 2**

Title: Set interrupt mode 2 condition

Description: *See manufacturer's data*

Mnemonic: **IN r, (C)**

Title: Load register from port (C)

Description: *See manufacturer's data*

Mnemonic: **IN A, (N)**

Title: Load accumulator from input port N

Description: *See manufacturer's data*

Mnemonic: **INC r**

Title: Increase register r

Symbolic: r←r + 1

Description: Specified register contents are increased by one

Flags:

S	Z	H	P/V	N	C
*	*	*	*	0	

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	INC A	3C	1
	INC B	04	1
	INC C	0C	1
	INC D	14	1
	INC E	1C	1
	INC H	24	1
	INC L	2C	1

Mnemonic: **INC rr**

Title: Increment register pair

Symbolic: rr←rr + 1

Description: Specified register pair contents are increased by one

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	INC BC	03	1
	INC DE	13	1
	INC HL	23	1
	INC SP	33	1

Mnemonic: **INC (HL)**

Title: Increment indirectly addressed location (HL)

Symbolic: (HL)←(HL) + 1

Description: The contents of the byte addressed by HL are increased by one

Flags:

S	Z	H	P/V	N	C
*	*	*	*	0	

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	INC (HL)	34	1

Mnemonic: **INC (IX + d)**

Title: Increment indexed address location (IX + d)

Symbolic: (IX + d)←(IX + d) + 1

Description: The contents of the indexed addressed byte are increased by one

Flags:

S	Z	H	P/V	N	C
*	*	*	*	0	

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indexed	INC (IX + d)	DD 34 d	3

Mnemonic: **INC (IY + d)**

Title: Increment indexed address location (IY + d)

Symbolic: (IY + d)←(IY + d) + 1

Description: The contents of the indexed addressed byte are increased by one

Flags:

S	Z	H	P/V	N	C
*	*	*	*	0	

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indexed	INC (IY + d)	FD 34 d	3

Mnemonic: **INC IX**
Title: Increment IX
Symbolic: IX←IX + 1
Description: The contents of the IX register are increased by one
Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	INC IX	DD 23	2

Mnemonic: **INC IY**
Title: Increment IY
Symbolic: IY←IY + 1
Description: The contents of the IY register are increased by one
Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	INC IY	FD 23	2

Mnemonic: **IND**
Title: Input with decrement
Description: *See manufacturer's data*

Mnemonic: **INDR**
Title: Block input with decrement
Description: *See manufacturer's data*

Mnemonic: **INI**
Title: Input with increment
Description: *See manufacturer's data*

Mnemonic: **INIR**

Title: Block input with increment

Description: *See manufacturer's data*

Mnemonic: **JP cc, pq**

Title: Conditional jump to address pq

Symbolic: If condition true then PC←pq

Description: If the condition is met then the address pq is placed into the program counter register. This results in a jump to the specified address

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Immediate	JP NZ, pq	C2 q p	3
	JP Z, pq	CA q p	3
	JP NC, pq	D2 q p	3
	JP C, pq	DA q p	3
	JP PO, pq	E2 q p	3
	JP PE, pq	EA q p	3
	JP P, pq	F2 q p	3
	JP M, pq	FA q p	3

Mnemonic: **JP pq**

Title: Jump to location pq

Symbolic: PC←pq

Description: The address pq is placed into the program counter register. This results in a jump to the specified address

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Immediate	JP pq	C3 q p	3

Mnemonic:	**JP (HL)**
Title:	Jump to HL
Symbolic:	PC←HL
Description:	The contents of HL are placed in the program counter. This results in a jump to the address specified by HL
Flags:	*No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	JP (HL)	E9	1

Mnemonic:	**JP (IX)**
Title:	Jump to IX
Symbolic:	PC←HL
Description:	The contents of IX are placed in the program counter. This results in a jump to the address specified by IX
Flags:	*No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	JP IX	DD E9	2

Mnemonic:	**JP (IY)**
Title:	Jump to IY
Symbolic:	PC←IY
Description:	The contents of IY are placed in the program counter. This results in a jump to the address specified by IY
Flags:	*No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	JP (IY)	FD E9	2

Mnemonic: **JR cc, e**

Title: Conditional relative jump

Symbolic: If condition is true then PC←PC + e

Description: A relative jump is performed if the given condition is met

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Relative	JR NZ, e	20 e	2
	JR Z, e	28 e	2
	JR NC, e	30 e	2
	JR C, e	38 e	2

Mnemonic: **JR e**

Title: Unconditional relative jump

Symbolic: PC←PC + e

Description: The given offset e is added to the program counter using two's complement arithmetic (See chapter 8).

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Relative	JR e	18 e	2

Mnemonic: **LD rr, (pq)**

Title: Load register pair from locations addressed by pq

Symbolic: rr low←(pq)
rr high←(pq + 1)

Description: The register pair is loaded with the contents of byte pq and byte pq + 1

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Extended absolute	LD BC, (pq)	ED 4B q p	4
	LD DE, (pq)	ED 5B q p	4
	LD HL, (pq)	ED 6B q p	4
	LD SP, (pq)	ED 7B q p	4

Mnemonic:	**LD rr, mn**
Title:	Load register pair with immediate data
Symbolic:	rr←mn
Description:	The contents of the two memory locations immediately following the op code are placed into the register pair
Flags:	*No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Immediate	LD BC, mn	01 n m	3
	LD DE, mn	11 n m	3
	LD HL, mn	21 n m	3
	LD SP, mn	31 n m	3

Mnemonic:	**LD r, n**
Title:	Load register r with immediate data n
Symbolic:	r←n
Description:	The contents of the byte following the op code is loaded into the specified register
Flags:	*No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Immediate	LD A, n	3E n	2
	LD B, n	06 n	2
	LD C, n	0E n	2
	LD D, n	16 n	2
	LD E, n	1E n	2
	LD H, n	26 n	2
	LD L, n	2E n	2

Mnemonic:	**LD r, r1**
Title:	Load register r from register r1
Symbolic:	r←r1
Description:	The contents of the source register r1 are loaded into register r
Flags:	*No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	LD A, A	7F	1
	LD A, B	78	1

LD A, C	79	1
LD A, D	7A	1
LD A, E	7B	1
LD A, H	7C	1
LD A, L	7D	1
LD B, A	47	1
LD B, B	40	1
LD B, C	41	1
LD B, D	42	1
LD B, E	43	1
LD B, H	44	1
LD B, L	45	1
LD C, A	4F	1
LD C, B	48	1
LD C, C	49	1
LD C, D	4A	1
LD C, E	4B	1
LD C, H	4C	1
LD C, L	4D	1
LD D, A	57	1
LD D, B	50	1
LD D, C	51	1
LD D, D	52	1
LD D, E	53	1
LD D, H	54	1
LD D, L	55	1
LD E, A	5F	1
LD E, B	58	1
LD E, C	59	1
LD E, D	5A	1
LD E, E	5B	1
LD E, H	5C	1
LD E, L	5D	1
LD H, A	67	1
LD H, B	60	1
LD H, C	61	1
LD H, D	62	1
LD H, E	63	1
LD H, H	64	1
LD H, L	65	1
LD L, A	6F	1
LD L, B	68	1
LD L, C	69	1
LD L, D	6A	1
LD L, E	6B	1
LD L, H	6C	1
LD L, L	6D	1

Mnemonic: **LD (BC), A**

Title: Load indirectly addressed location (BC) from accumulator

Symbolic: (BC)←A

Description: The contents of the byte addressed by BC are loaded with the contents of the accumulator

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	LD (BC), A	02	1

Mnemonic: **LD (DE), A**

Title: Load indirectly addressed location (DE) from accumulator

Symbolic: (DE)←A

Description: The contents of the byte addressed by DE are loaded with the contents of the accumulator

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	LD (DE), A	12	1

Mnemonic: **LD (HL), n**

Title: Load indirect addressed location (HL) with immediate data byte

Symbolic: (HL)←n

Description: The contents of the byte following the op code are loaded into the indirectly addressed location (HL)

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Immediate Indirect	LD (HL), n	36 n	2

Mnemonic: **LD (HL), r**

Title: Load indirect addressed location (HL) from specified register r

Symbolic: (HL)←r

Description: The contents of the specified register are loaded into the memory location whose address is in HL

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	LD (HL), A	77	1
	LD (HL), B	70	1
	LD (HL), C	71	1
	LD (HL), D	72	1
	LD (HL), E	73	1
	LD (HL), H	74	1
	LD (HL), L	75	1

Mnemonic: **LD r, (IX + d)**

Title: Load register r from location (IX + d)

Symbolic: r←(IX + d)

Description: The contents of the byte with address (IX + d) is placed into register r

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indexed	LD A, (IX + d)	DD 7E d	3
	LD B, (IX + d)	DD 46 d	3
	LD C, (IX + d)	DD 4E d	3
	LD D, (IX + d)	DD 56 d	3
	LD E, (IX + d)	DD 5E d	3
	LD H, (IX + d)	DD 66 d	3
	LD L, (IX + d)	DD 6E d	3

Mnemonic: **LD r, (IY + d)**

Title: Load register r from location (IY + d)

Symbolic: r←(IY + d)

Description: The contents of the byte with address (IY + d) is placed into register r

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indexed	LD A, (IY + d)	FD 7E d	3
	LD B, (IY + d)	FD 46 d	3
	LD C, (IY + d)	FD 4E d	3
	LD D, (IY + d)	FD 56 d	3
	LD E, (IY + d)	FD 5E d	3
	LD H, (IY + d)	FD 66 d	3
	LD L, (IY + d)	FD 6E d	3

Mnemonic: **LD (IX + d), n**

Title: Load indexed addressed location (IX + d) with immediate data byte

Symbolic: (IX + d)←n

Description: The contents of the byte following the op code are placed into the byte whose address is specified by (IX + d)

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indexed Immediate	LD (IX + d), n	DD 36 d n	4

Mnemonic: **LD (IY + d), n**

Title: Load indexed addressed location (IY + d) with immediate data byte

Symbolic: (IY + d)←n

Description: The contents of the byte following the op code are placed into the byte whose address is specified by (IY + d)

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indexed Immediate	LD (IY + d), n	FD 36 d n	4

Mnemonic: **LD (IX + d), r**

Title: Load indexed address location (IX + d) from specified register r

Symbolic: (IX + d)←r

Description: The contents of register r are transferred to the location specified by (IX + d)

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indexed	LD (IX + d), A	DD 77 d	3
	LD (IX + d), B	DD 70 d	3
	LD (IX + d), C	DD 71 d	3
	LD (IX + d), D	DD 72 d	3
	LD (IX + d), E	DD 73 d	3
	LD (IX + d), H	DD 74 d	3
	LD (IX + d), L	DD 75 d	3

Mnemonic: **LD (IY + d), r**

Title: Load indexed address location (IY + d) from specified register r

Symbolic: (IY + d)←r

Description: The contents of register r are transferred to location specified by (IY + d)

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indexed	LD (IY + d), A	FD 77 d	3
	LD (IY + d), B	FD 70 d	3
	LD (IY + d), C	FD 71 d	3
	LD (IY + d), D	FD 72 d	3
	LD (IY + d), E	FD 73 d	3
	LD (IY + d), H	FD 74 d	3
	LD (IY + d), L	FD 75 d	3

Mnemonic:	**LD A, (pq)**
Title:	Load accumulator from location whose address is given by pq
Symbolic:	A←(pq)
Description:	The contents of the location addressed by pq are placed into the accumulator
Flags:	No effect

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Extended absolute	LD A, (pq)	3A q p	3

Mnemonic:	**LD (pq), A**
Title:	Load directly addressed memory (pq) from accumulator
Symbolic:	(pq)←A
Description:	The contents of the accumulator are placed in location pq
Flags:	No effect

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Extended absolute	LD (pq), A	32 q p	3

Mnemonic:	**LD (pq), rr**
Title:	Load memory locations addressed by pq from a register pair rr
Symbolic:	(pq)←rr low (pq + 1)←rr high
Description:	The contents of the specified registers are placed into locations whose addresses are pq and pq + 1
Flags:	No effect

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Extended absolute	LD (pq), BC	ED 43 q p	4
	LD (pq), DE	ED 53 q p	4
	LD (pq), HL	ED 63 q p	4
	LD (pq), SP	ED 73 q p	4

Mnemonic: **LD (pq), HL**

Title: Load locations addressed by pq from HL

Symbolic: (pq)←L
(pq + 1)←H

Description: The contents of H and L are placed into locations pq + 1 and pq respectively

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Extended absolute	LD (pq), HL	22 q p	3

Mnemonic: **LD (pq), IX**

Title: Load locations addressed by pq from IX

Symbolic: (pq)←IX low
(pq + 1)←IX high

Description: The contents of IX are placed into locations pq + 1 and pq respectively

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Extended absolute	LD (pq), IX	DD 22 q p	4

Mnemonic: **LD (pq), IY**

Title: Load locations addressed by pq from IY

Symbolic: (pq)←IY low
(pq + 1)←IY high

Description: The contents of IY are placed into locations pq + 1 and pq respectively

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Extended absolute	LD (pq), IY	FD 22 q p	4

Mnemonic: **LD A, (BC)**

Title: Load accumulator from indirectly addressed location given by BC

Symbolic: A←(BC)

Description: The contents of the byte whose address is in BC are loaded into the accumulator

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	LD A, (BC)	0A	1

Mnemonic: **LD A, (DE)**

Title: Load accumulator from indirectly addressed location given by DE

Symbolic: A←(DE)

Description: The contents of the byte whose address is in DE are loaded into the accumulator

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	LD A, (DE)	1A	1

Mnemonic: **LD A, I**

Title: Load accumulator from the interrupt vector register

Description: *See manufacturer's data*

Mnemonic: **LD A, R**

Title: Load accumulator from memory refresh register

Description: *See manufacturer's data*

Mnemonic: **LD I, A**

Title: Load interrupt vector register from accumulator

Description: *See manufacturer's data*

Mnemonic: **LD HL, (pq)**

Title: Load HL from locations addressed by pq

Symbolic: L←(pq)
H←(pq + 1)

Description: The contents of bytes addressed by pq and pq + 1 are loaded into the HL pair

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Extended absolute	LD HL, (pq)	2A q p	3

Mnemonic: **LD IX, mn**

Title: Load register IX with immediate data mn

Symbolic: IX←mn

Description: The contents of the two bytes following the op code are loaded into the IX register.

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Immediate	LD IX, mn	DD 21 n m	4

Mnemonic: **LD IX, (pq)**

Title: Load IX register from location with address pq

Symbolic: IX low←(pq)
IX high←(pq + 1)

Description: Register IX is loaded with the contents of the bytes with address pq and pq + 1

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Extended absolute	LD IX, (pq)	DD 2A q p	4

Mnemonic: **LD IY, mn**

Title: Load register IY with immediate data mn

Symbolic: IY←mn

Description: The contents of the two bytes following the op code are loaded into the IY register.

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Immediate	LD IY, mn	FD 21 n m	4

Mnemonic: **LD IY, (pq)**

Title: Load IY register from location with address pq

Symbolic: IY low←(pq)
IY high←(pq + 1)

Description: Register IY is loaded with the contents of the bytes with address pq and pq + 1

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Extended absolute	LD IY, (pq)	FD 2A q p	4

Mnemonic: **LD R, A**

Title: Load refresh register from accumulator

Description: *See manufacturer's data*

Mnemonic: **LD SP, HL**

Title: Load stack pointer from HL register pair

Symbolic: SP←HL

Description: The contents of the HL register pair are copied into the stack pointer register

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	LD SP, HL	F9	1

Mnemonic:	**LD SP, IX**
Title:	Load stack pointer from index register IX
Symbolic:	SP←IX
Description:	The contents of the IX register are copied into the stack pointer register
Flags:	*No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	LD SP, IX	DD F9	2

Mnemonic:	**LD SP, IY**
Title:	Load stack pointer from index register IY
Symbolic:	SP←IY
Description:	The contents of the IY register are copied into the stack pointer register
Flags:	*No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	LD SP, IY	FD F9	2

Mnemonic:	**LDD**
Title:	Block load with decrement
Symbolic:	(DE)←(HL) : DE←DE – 1 : HL←HL – 1 : BC←BC – 1
Description:	The contents of the byte addresssed by HL are loaded into the location addressed by DE. Then HL, DE and the BC pair are each decremented

Flags:

S	Z	H	P/V	N	C
		0	?	0	

P/V flag reset if BC=0 after execution, else set

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	LDD	ED A8	2

Mnemonic: **LDDR**

Title: Repeating block load with decrement

Symbolic: (DE)←(HL) : DE←DE – 1
: HL←HL – 1
: BC←BC – 1
Repeated until BC=0

Description: The contents of byte addresssed by HL are loaded into the location addressed by DE. Then HL, DE and the BC pair are each decremented. If BC does *not* equal zero then program counter is decreased by 2 and the instruction repeated

Flags:

S	Z	H	P/V	N	C
		0	0	0	

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	LDDR	ED B8	2

Mnemonic: **LDI**

Title: Block load with increment

Symbolic: (DE)←(HL) : DE←DE + 1
: HL←HL + 1
: BC←BC – 1

Description: The contents of the byte addresssed by HL are loaded into the location addressed by DE. Then HL and DE are incremented and the BC pair decremented

Flags:

S	Z	H	P/V	N	C
		0	?	0	

P/V flag reset if BC=0 after execution, else set

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	LDI	ED A0	2

Mnemonic: **LDIR**

Title: Repeating block load with increment

Symbolic: (DE)←(HL) : DE←DE + 1
: HL←HL + 1
: BC←BC – 1
Repeated until BC=0

Description: The contents of the byte addresssed by HL are loaded into the location addressed by DE. Then HL and DE are incremented and the BC pair decremented. If BC does *not* equal zero then program counter is decreased by 2 and the instruction repeated

Flags:

S	Z	H	P/V	N	C
		0	0	0	

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	LDIR	ED B0	2

Mnemonic: **LD r, (HL)**

Title: Load register r from byte addressed by HL

Symbolic: r←(HL)

Description: The contents of the byte addressed by HL are loaded into register r

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	LD A, (HL)	7E	1
	LD B, (HL)	46	1
	LD C, (HL)	4E	1
	LD D, (HL)	56	1
	LD E, (HL)	5E	1
	LD H, (HL)	66	1
	LD L, (HL)	6E	1

Mnemonic: **NEG**

Title: Negate accumulator

Symbolic: A←0 – A

Description: The contents of the accumulator are subtracted from zero using two's complement arithmetic and result placed into accumulator

Flags:

S	Z	H	P/V	N	C
*	*	*	*	1	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	NEG	ED 44	2

Mnemonic: **NOP**

Title: No operation

Description: Performs no function other than to create a delay

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicict	NOP	00	1

Mnemonic: **OR s**

Title: Logical OR of accumulator with specified operand

Symbolic: A← – A ∨ s

Description: The contents of the accumulator are OR'ed with specified byte and the result placed into the accumulator. The rule table for OR is as follows:

	Accumulator bit	0	1
Operand bit	0	0	1
	1	1	1

Result in accumulator

Flags:

S	Z	H	P/V	N	C
*	*	0	*	0	0

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	OR A	B7	1
	OR B	B0	1
	OR C	B1	1
	OR D	B2	1
	OR E	B3	1
	OR H	B4	1
	OR L	B5	1
Immediate	OR n	F6 n	2
Indirect	OR, (HL)	B6	1
Indexed	OR, (IX + d)	DD B6 d	3
	OR, (IY + d)	FD B6 d	3

Mnemonic: **OTDR**

Title: Block output with decrement

Description: *See manufacturer's data*

Mnemonic: **OTIR**

Title: Block output with increment

Description: *See manufacturer's data*

Mnemonic: **OUT (C), r**

Title: Output register C to output port

Description: *See manufacturer's data*

Mnemonic: **OUT (N), A**

Title: Output accumulator to port N

Description: *See manufacturer's data*

Mnemonic: **OUTD**

Title: Output with decrement

Description: *See manufacturer's data*

Mnemonic: **OUTI**

Title: Output with increment

Description: *See manufacturer's data*

Mnemonic: **POP rr**

Title: Pop register pair from stack

Description: Current two bytes at top of stack are copied into the specified register pair

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	POP BC	C1	1
	POP DE	D1	1
	POP HL	E1	1
	POP AF	F1	1

Mnemonic: **POP IX**

Title: POP IX register from top of stack

Description: The contents of the top of stack are placed into the IX register

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	POP IX	DD E1	2

Mnemonic: **POP IY**

Title: POP IY register from top of stack

Description: The contents of the top of stack are placed into the IY register

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	POP IY	FD E1	2

Mnemonic: **PUSH rr**

Title: Push register pair onto the stack

Description: Specified register pair are copied to the top of the stack

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	PUSH BC	C5	1
	PUSH DE	D5	1
	PUSH HL	E5	1
	PUSH AF	F5	1

Mnemonic: **PUSH IX**

Title: Push IX onto stack

Description: The contents of the IX register are copied onto the stack

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	PUSH IX	DD E5	2

Mnemonic: **PUSH IY**

Title: Push IY onto stack

Description: The contents of the IY register are copied onto the stack

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	PUSH IY	FD E5	2

Mnemonic: **RES b, (HL)**

Title: Reset bit b of indirectly addressed operand

Description: Specified bit b of the byte whose address in contained in HL is set to zero

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	RES 0, (HL)	CB 86	2
	RES 1, (HL)	CB 8E	2
	RES 2, (HL)	CB 96	2
	RES 3, (HL)	CB 9E	2
	RES 4, (HL)	CB A6	2
	RES 5, (HL)	CB AE	2
	RES 6, (HL)	CB B6	2
	RES 7, (HL)	CB BE	2

Mnemonic: **RES b, (IX + d)**

Title: Reset bit b of IX indexed addressed operand

Description: Specified bit b of the byte specified by (IX + d) is set to zero

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indexed	RES 0, (IX + d)	DD CB d 86	4
	RES 1, (IX + d)	DD CB d 8E	4
	RES 2, (IX + d)	DD CB d 96	4
	RES 3, (IX + d)	DD CB d 9E	4
	RES 4, (IX + d)	DD CB d A6	4
	RES 5, (IX + d)	DD CB d AE	4
	RES 6, (IX + d)	DD CB d B6	4
	RES 7, (IX + d)	DD CB d BE	4

Mnemonic: **RES b, (IY + d)**

Title: Reset bit b of IY indexed addressed operand

Description: Specified bit b of the byte specified by (IY + d) is set to zero

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indexed	RES 0, (IY + d)	FD CB d 86	4
	RES 1, (IY + d)	FD CB d 8E	4
	RES 2, (IY + d)	FD CB d 96	4
	RES 3, (IY + d)	FD CB d 9E	4
	RES 4, (IY + d)	FD CB d A6	4
	RES 5, (IY + d)	FD CB d AE	4
	RES 6, (IY + d)	FD CB d B6	4
	RES 7, (IY + d)	FD CB d BE	4

Mnemonic: **RES b, r**

Title: Reset bit b of register r

Description: Specified bit b of register r is set to zero

Note: This implied addressing instruction is two bytes long. The first byte is CB hex, the second byte depends on which register and which bit is being reset. The following table shows the second byte op code possibilities:

REGISTERS		A	B	C	D	E	H	L
	0	87	80	81	82	83	84	85
	1	8F	88	89	8A	8B	8C	8D
B	2	97	90	91	92	93	94	95
I	3	9F	98	99	9A	9B	9C	9D
T	4	A7	A0	A1	A2	A3	A4	A5
S	5	AF	A8	A9	AA	AB	AC	AD
	6	B7	B0	B1	B2	B3	B4	B5
	7	BF	B8	B9	BA	BB	BC	BD

Mnemonic: **RET**

Title: Return from subroutine

Description: Two bytes currently at the top of the stack are copied into the program counter, this provides the address for the next instruction

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	RET	C9	1

Mnemonic: **RET cc**

Title: Conditional return from subroutine

Description: As for RET but only executed if condition is met

Flags: *No effect*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	RET NZ	C0	1
	RET Z	C8	1
	RET NC	D0	1
	RET C	D8	1
	RET PO	E0	1
	RET PE	E8	1
	RET P	F0	1
	RET M	F8	1

Mnemonic: **RETI**

Title: Return from interrupt

Description: *See manufacturer's data*

Mnemonic: **RETN**

Title: Return from non-maskable interrupt

Description: *See manufacturer's data*

Mnemonic: **RL s**

Title: Rotate operand s left through carry

Description: The contents of the specified operand are left shifted one place. Original carry flag contents are moved to bit 0 and original bit 7 contents moved to the carry flag. The instruction thus performs a 9-bit rotation.

Flags:

S	Z	H	P/V	N	C
*	*	0	*	0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	RL A	CB 17	2
	RL B	CB 10	2
	RL C	CB 11	2
	RL D	CB 12	2
	RL E	CB 13	2
	RL H	CB 14	2
	RL L	CB 15	2
Indirect	RL (HL)	CB 16	2
Indexed	RL (IX + d)	DD CB d 16	4
	RL (IY + d)	FD CB d 16	4

Mnemonic: **RLA**

Title: Rotate accumulator left through carry

Description: The contents of the accumulator are shifted left. The contents of carry flag are moved to bit 0 and contents of bit 7 are moved into carry. (ie. this is a 9-bit rotation)

Flags:

S	Z	H	P/V	N	C
		0		0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	RLA	17	1

Mnemonic: **RLCA**

Title: Rotate accumulator left with branch carry

Description: *See manufacturer's data*

Mnemonic: **RLC r**

Title: Rotate register r left with branch carry

Description: *See manufacturer's data*

Mnemonic: **RLC (HL)**

Title: Rotate byte addressed by HL left with branch carry

Description: *See manufacturer's data*

Mnemonic: **RLC (IX + d)**

Title: Rotate left with branch carry the location addressed by (IX + d)

Description: *See manufacturer's data*

Mnemonic: **RLC (IY + d)**

Title: Rotate left with branch carry the location addressed by (IY + d)

Description: *See manufacturer's data*

Mnemonic: **RLD**

Title: Rotate left decimal

Description: *See manufacturer's data*

Mnemonic:	**RR s**
Title:	Rotate operand s right through carry
Description:	The contents of the carry flag plus specified byte are treated as a 9-bit word for rotation to the right

Flags:

S	Z	H	P/V	N	C
*	*	0	*	0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	RR A	CB 1F	2
	RR B	CB 18	2
	RR C	CB 19	2
	RR D	CB 1A	2
	RR E	CB 1B	2
	RR H	CB 1C	2
	RR L	CB 1D	2
Indirect	RR (HL)	CB 1E	2
Indexed	RR (IX + d)	DD CB d 1E	4
	RR (IY + d)	FD CB d 1E	4

Mnemonic:	**RRA**
Title:	Rotate accumulator right through carry
Description:	The contents of the carry flag together with the specified operand are treated as a 9-bit word for a right rotation

Flags:

S	Z	H	P/V	N	C
		0		0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	RRA	1F	1

Mnemonic:	**RRC s**
Title:	Rotate right with branch carry
Description:	*See manufacturer's data*

Mnemonic:	**RRCA**
Title:	Rotate accumulator right with branch carry
Description:	*See manufacturer's data*

Mnemonic: **RRD**

Title: Rotate right decimal

Description: *See manufacturer's data*

Mnemonic: **RST p**

Title: Restart at p

Description: *See manufacturer's data*

Mnemonic: **SBC A, s**

Title: Subtract with borrow

Symbolic: A←A – s – Carry

Description: The specified operand, together with the carry flag, is subtracted from the accumulator

Flags:

S	Z	H	P/V	N	C
*	*	*	*	1	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	SBC A, A	9F	1
	SBC A, B	98	1
	SBC A, C	99	1
	SBC A, D	9A	1
	SBC A, E	9B	1
	SBC A, H	9C	1
	SBC A, L	9D	1
Immediate	SBC A, n	DE n	1
Indirect	SBC A, (HL)	9E	1
Indexed	SBC A, (IX + d)	DD 9E d	3
	SBC A, (IY + d)	FD 9E d	3

Mnemonic: **SBC HL, rr**

Title: Subtract register pair with borrow from HL

Symbolic: HL←HL – rr – Carry

Description: The contents of the register pair, together with the carry, are subtracted from contents of HL. Result is stored in HL

Flags:

S	Z	H	P/V	N	C
*	*	*	*	1	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	SBC HL, BC	ED 42	2
	SBC HL, DE	ED 52	2
	SBC HL, HL	ED 62	2
	SBC HL, SP	ED 72	2

Mnemonic: **SCF**

Title: Set carry flag

Symbolic: Carry←1

Description: The carry flag is set to 1

Flags:

S	Z	H	P/V	N	C
		0		0	1

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	SCF	37	1

Mnemonic: **SET b, (HL)**

Title: Set bit b of indirectly addressed operand

Description: Bit b of the byte whose address is contained in HL is set to 1

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indirect	SET 0, (HL)	CB C6	2
	SET 1, (HL)	CB CE	2
	SET 2, (HL)	CB D6	2
	SET 3, (HL)	CB DE	2
	SET 4, (HL)	CB E6	2
	SET 5, (HL)	CB EE	2
	SET 6, (HL)	CB F6	2
	SET 7, (HL)	CB FE	2

Mnemonic:	**SET b, (IX + d)**
Title:	Set bit b of IX indexed addressed operand
Description:	Bit b of the byte whose address is specified by (IX + d) is set to 1

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indexed	SET 0, (IX + d)	DD CB d C6	4
	SET 1, (IX + d)	DD CB d CE	4
	SET 2, (IX + d)	DD CB d D6	4
	SET 3, (IX + d)	DD CB d DE	4
	SET 4, (IX + d)	DD CB d E6	4
	SET 5, (IX + d)	DD CB d EE	4
	SET 6, (IX + d)	DD CB d F6	4
	SET 7, (IX + d)	DD CB d FE	4

Mnemonic:	**SET b, (IY + d)**
Title:	Set bit b of IY indexed addressed operand
Description:	Bit b of the byte whose address is specified by (IY + d) is set to 1

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Indexed	SET 0, (IY + d)	FD CB d C6	4
	SET 1, (IY + d)	FD CB d CE	4
	SET 2, (IY + d)	FD CB d D6	4
	SET 3, (IY + d)	FD CB d DE	4
	SET 4, (IY + d)	FD CB d E6	4
	SET 5, (IY + d)	FD CB d EE	4
	SET 6, (IY + d)	FD CB d F6	4
	SET 7, (IY + d)	FD CB d FE	4

Mnemonic: **SET b, r**

Title: Set bit b of register r

Description: The selected bit of register r is set to 1

Note: This implied addressing instruction is two bytes long. The first byte is CB hex, the second byte depends on which register and which bit is being set. The following table shows the second byte op code possibilities:

	REGISTERS	A	B	C	D	E	H	L
	0	C7	C0	C1	C2	C3	C4	C5
	1	CF	C8	C9	CA	CB	CC	CD
B	2	D7	D0	D1	D2	D3	D4	D5
I	3	DF	D8	D9	DA	DB	DC	DD
T	4	E7	E0	E1	E2	E3	E4	E5
S	5	EF	E8	E9	EA	EB	EC	ED
	6	F7	F0	F1	F2	F3	F4	F5
	7	FF	F8	F9	FA	FB	FC	FD

Mnemonic: **SLA s**

Title: Arithmetic shift left

Description: The contents of the specified operand are shifted to the left. Bit 0 is set to 0 and bit 7 transferred to the carry.

Flags:

S	Z	H	P/V	N	C
*	*	0	*	0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	SLA A	CB 27	2
	SLA B	CB 20	2
	SLA C	CB 21	2
	SLA D	CB 22	2
	SLA E	CB 23	2
	SLA H	CB 24	2
	SLA L	CB 25	2
Indirect	SLA (HL)	CB 26	2
Indexed	SLA (IX + d)	DD CB d 26	4
	SLA (IY + d)	FD CB d 26	4

Mnemonic: **SRA s**

Title: Arithmetic shift right

Description: The contents of the specified operand are shifted to the right. Bit 0 is transferred to the carry. The contents of bit 7 remain unchanged.

Flags:

S	Z	H	P/V	N	C
*	*	0	*	0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	SRA A	CB 2F	2
	SRA B	CB 28	2
	SRA C	CB 29	2
	SRA D	CB 2A	2
	SRA E	CB 2B	2
	SRA H	CB 2C	2
	SRA L	CB 2D	2
Indirect	SRA (HL)	CB 2E	2
Indexed	SRA (IX + d)	DD CB d 2E	4
	SRA (IY + d)	FD CB d 2E	4

Mnemonic: **SRL s**

Title: Logical shift right

Description: This is a non-rotation shift. All bits are shifted right by one position. Bit 7 is set to zero and bit 0 is moved into the carry

Flags:

S	Z	H	P/V	N	C
*	*	0	*	0	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	SRL A	3F	1
	SRL B	38	1
	SRL C	39	1
	SRL D	3A	1
	SRL E	3B	1
	SRL H	3C	1
	SRL L	3D	1
Indirect	SRL (HL)	CB 3E	2
Indexed	SRL (IX + d)	DD CB d 3E	4
	SRL (IY + d)	FD CB d 3E	4

Mnemonic: **SUB s**

Title: Subtract operand from accumulator

Symbolic: A←A – s

Description: The specified operand is subtracted from the contents of the accumulator. The result is placed into the accumulator

Flags:

S	Z	H	P/V	N	C
*	*	*	*	1	*

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	SUB A	97	1
	SUB B	90	1
	SUB C	91	1
	SUB D	92	1
	SUB E	93	1
	SUB H	94	1
	SUB L	95	1
Immediate	SUB n	D6 n	2
Indirect	SUB (HL)	96	1
Indexed	SUB (IX + d)	DD 96 d	3
	SUB (IY + d)	FD 96 d	3

Mnemonic: **XOR s**

Title: Exclusive OR accumulator with specified operand

Description: The accumulator and the operand are compared bit by bit according to the following rule table:

```
Accumulator Bit  0   1
                ┌──────
Operand      0  │ 0   1
Bit             │            Result in accumulator
             1  │ 1   0
```

Flags:

S	Z	H	P/V	N	C
*	*	0	*	0	0

Addressing mode	*Assembly form*	*Object code*	*Bytes*
Implicit	XOR A	AF	1
	XOR B	A8	1
	XOR C	A9	1
	XOR D	AA	1
	XOR E	AB	1
	XOR H	AC	1
	XOR L	AD	1
Immediate	XOR n	EE n	2
Indirect	XOR (HL)	AE	1
Indexed	XOR (IX + d)	DD AE d	3
	XOR (IY + d)	FD AE d	3

Appendix B
ASSEMBLER CONVENTIONS

Just as various implementations of the BASIC language often differ from one another, so different assemblers also vary in their syntax requirements and in the facilities available. For the purposes of standardization we have adopted the following conventions for all Z-80 source code. These conventions are used in the examples in this book.

- All numbers must start with a digit, 0-9, and are classed as *decimal* numbers unless:

  They end in 'H' in which case they are hexadecimal.

  They end in 'B' in which case they are binary.

  They end in 'Q' in which case they are octal.

Thus `12`, `0BBFH`, `77Q` and `01100010B` are all examples of valid numbers.

- Labels must end in a colon but can be any number of characters long. The dollar character (`$`) can be used within a label to improve the clarity of the label. Thus `STACK:`, `CR$LF:` and `OUTPUT$ROUTINE:` are all examples of valid labels.

- Remarks within the source are of two types:

  Any line beginning with an asterisk, (*), is treated as a whole line remark. It is ignored by the assembler completely.

  Anything written after a semi-colon (;) delimiter will also be ignored by the assembler.

- All standard Zilog mnemonics are supported. This may seem a strange convention, but it is not uncommon to find Z-80 assemblers which use 8080 mnemonics for Z-80-compatible 8080 instructions. The translation to 8080 mnemonics is only one of direct translation and should not cause problems (for example, the Z-80 instruction LD C, 0 has an 8080 equivalent of MVI C,0).

- The following directives are used to reserve space within a program:

  DS reserves uninitialized space.

DB reserves initialized space. This may be a valid single byte number, a label equating to a single byte value or a string of characters assembled into consecutive locations. The statement:

```
DB   12, 0FFH, SPACE, 'Out of memory$'
```

will place 12 into the first byte, FF_{hex} into the second byte, 32 into the third byte (SPACE must have been previously defined as EQU to 32), and then the ASCII codes corresponding to the string 'Out of memory$'.

DW defines a two byte 'word' format. The value given is converted to a 16-bit value and stored *low byte first* as is the standard Z-80 convention. As an example the statement DW SPACE would place 32 into the low byte and 0 into the high byte.

- The statement ORG defines a starting location for assembly.

Appendix C
ASCII CHARACTER SET

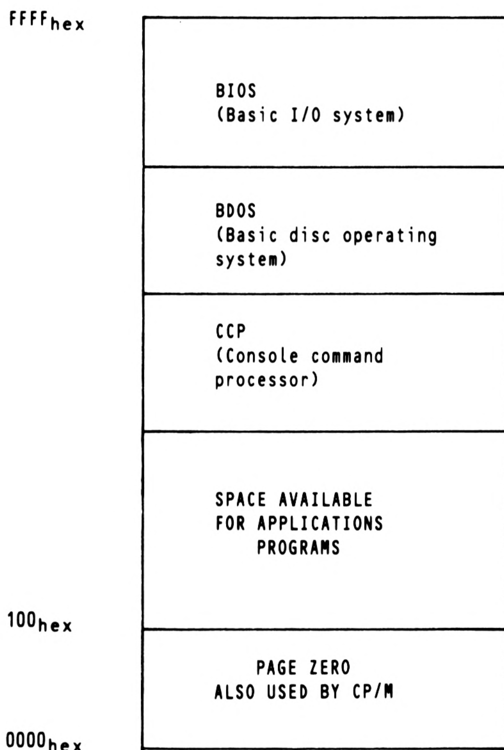
Decimal	Hex	Binary	ASCII
00	00	000 0000	NUL
01	01	000 0001	SOH
02	02	000 0010	STX
03	03	000 0011	ETX
04	04	000 0100	EOT
05	05	000 0101	ENQ
06	06	000 0110	ACK
07	07	000 0111	BEL
08	08	000 1000	BS
09	09	000 1001	HT
10	0A	000 1010	LF
11	0B	000 1011	VT
12	0C	000 1100	FF
13	0D	000 1101	CR
14	0E	000 1110	SO
15	0F	000 1111	SI
16	10	001 0000	DLE
17	11	001 0001	DC1
18	12	001 0010	DC2
19	13	001 0011	DC3
20	14	001 0100	DC4
21	15	001 0101	NAK
22	16	001 0110	SYN
23	17	001 0111	ETB
24	18	001 1000	CAN
25	19	001 1001	EM
26	1A	001 1010	SUB
27	1B	001 1011	ESC
28	1C	001 1100	FS
29	1D	001 1101	GS
30	1E	001 1110	RS
31	1F	001 1111	US
32	20	010 0000	SP
33	21	010 0001	!
34	22	010 0010	"
35	23	010 0011	#
36	24	010 0100	$
37	25	010 0101	%
38	26	010 0110	&
39	27	010 0111	'
40	28	010 1000	(
41	29	010 1001	)
42	2A	010 1010	*

Decimal	Hex	Binary	ASCII
43	2B	010 1011	+
44	2C	010 1100	,
45	2D	010 1101	-
46	2E	010 1110	.
47	2F	010 1111	/
48	30	011 0000	0
49	31	011 0001	1
50	32	011 0010	2
51	33	011 0011	3
52	34	011 0100	4
53	35	011 0101	5
54	36	011 0110	6
55	37	011 0111	7
56	38	011 1000	8
57	39	011 1001	9
58	3A	011 1010	:
59	3B	011 1011	;
60	3C	011 1100	<
61	3D	011 1101	=
62	3E	011 1110	>
63	3F	011 1111	?
64	40	100 0000	@
65	41	100 0001	A
66	42	100 0010	B
67	43	100 0011	C
68	44	100 0100	D
69	45	100 0101	E
70	46	100 0110	F
71	47	100 0111	G
72	48	100 1000	H
73	49	100 1001	I
74	4A	100 1010	J
75	4B	100 1011	K
76	4C	100 1100	L
77	4D	100 1101	M
78	4E	100 1110	N
79	4F	100 1111	O
80	50	101 0000	P
81	51	101 0001	Q
82	52	101 0010	R
83	53	101 0011	S
84	54	101 0100	T
85	55	101 0101	U
86	56	101 0110	V
87	57	101 0111	W
88	58	101 1000	X
89	59	101 1001	Y
90	5A	101 1010	Z
91	5B	101 1011	[
92	5C	101 1100	\
93	5D	101 1101	]
94	5E	101 1110	↑
95	5F	101 1111	_
96	60	110 0000	'
97	61	110 0001	a
98	62	110 0010	b

Decimal	Hex	Binary	ASCII
99	63	110 0011	c
100	64	110 0100	d
101	65	110 0101	e
102	66	110 0110	f
103	67	110 0111	g
104	68	110 1000	h
105	69	110 1001	i
106	6A	110 1010	j
107	6B	110 1011	k
108	6C	110 1100	l
109	6D	110 1101	m
110	6E	110 1110	n
111	6F	110 1111	o
112	70	111 0000	p
113	71	111 0001	q
114	72	111 0010	r
115	73	111 0011	s
116	74	111 0100	t
117	75	111 0101	u
118	76	111 0110	v
119	77	111 0111	w
120	78	111 1000	x
121	79	111 1001	y
122	7A	111 1010	z
123	7B	111 1011	{
124	7C	111 1100	|
125	7D	111 1101	}
126	7E	111 1110	~
127	7F	111 1111	DEL

Appendix D
THE CP/M OPERATING SYSTEM

The following schematic diagram gives the logical memory layout for a standard CP/M based 64K computer.



Some systems may have the upper memory code 'moved down' to allow for such things as memory mapped display (the *Osborne*-01 is one machine on which this has been done). Occasionally other areas within the BDOS and BIOS may be modified or enhanced, but this does not usually affect the CP/M end user.

CP/M's functions are accessed through a common entry point at location 05_{hex}. Only a minimal understanding of the workings of CP/M is required for the programs given in this book and these are

as follows: All programs are expected to start at 100_{hex}. Space below this (ie. the zero page), is used by CP/M itself. The jump instruction that starts at location 05_{hex} can be used to determine the start of the disk operating system area (called the BDOS) and area from 100_{hex} up to the start of BDOS is available for use by applications programs. It is common practice to overwrite the CCP area since this is re-loaded when CP/M is re-booted.

CP/M system calls are performed by placing a 'function number' into the C register and then 'CALLing BDOS'. The only functions required for the programs in this book are those of input and output.

Function number = 1 (Console Input)

A typical use of the console input function is shown below:

```
* =====================================================================
*               CP/M -  C O N S O L E - I N P U T
* ---------------------------------------------------------------------
INPUT$ROUTINE: PUSH BC ! PUSH DE ! PUSH HL    ;Preserve registers
               LD   C,1                       ;Function number
               CALL BDOS                      ;CP/M entry point
               POP HL ! POP DE ! POP BC       ;Restore registers
               RET                            ;Return from subroutine
* =====================================================================
```

Function number = 2 (Console Output)

The character to be output must be present in the E register. Since the DE pair is commonly used by applications programs for other purposes it is common practice to write programs assuming that it is the accumulator that is used for I/O and then move the character from the accumulator to the E register from within the output routine. A typical example is shown below:

```
* =====================================================================
*            CP/M -  C O N S O L E - O U T P U T
* ---------------------------------------------------------------------
OUTPUT$ROUTINE: PUSH BC ! PUSH DE ! PUSH HL   ;Preserve registers
               LD   C,2                       ;Function number
               LD   E,A                       ;Transfer via E register
               CALL BDOS                      ;CP/M entry point
               POP HL ! POP DE ! POP BC       ;Restore registers
               RET                            ;Return from subroutine
* =====================================================================
```

Function number = 6 (Direct I/O)

The above CP/M calls recognize certain control characters as having special meanings. Occasionally such system interpretation is unwelcome and thus facilities are provided to allow for direct I/O without interpretation. If the E register is loaded with FF_{hex} then the function will collect characters from the console and return them in the accumulator. If no character is available then the accumulator is set to zero and thus a 'wait for input' loop must be used. When the E register is loaded with any character *other than* FF_{hex} then function 6 will output that character to the console. Typical examples of such routines are given in chapters 5 and 6.

NOTES

Appendix E
GLOSSARY

Algorithm
A series of rules (or a diagramatic equivalent) which when followed result in a predetermined or predictable outcome.

Alternation
A set of two or more alternative actions with only one of those actions being performed.

ASCII
American Standard Code for Information Interchange consists of a set of 96 displayable and 32 non-displayed characters based on a seven bit code.

Assembly Listing
A printed listing made by the assembler showing how the source code has been interpreted during the assembly process.

BCD
Binary Coded Decimal.

Binary
A number system using base 2 for its operations.

Binary Search
A method of searching an ordered table or file by successively dividing the search area in half.

Bit
A Binary Digit.

Boolean Algebra
The mathematics of a class of logical operations closely related to set theory mathematics.

BPS
Bits Per Second.

Buffer
An area of memory used to hold data temporarily whilst being collected or transmitted.

Bug

A fault within a program which has not yet been found. Also see *Debug*.

Chain

See *Linked list*.

Comment

A remark, social or otherwise, written within a program.

Complement

Binary complement, the process of turning all 1's to 0's and all 0's to 1's.

Computed addressing

A form of addressing in which the address is calculated at run time. Indirect addressing and indexed addressing are two examples.

Control character

A character which signifies the start or finish of some process.

CP/M

A commonly favoured operating system designed originally for the 8080 microprocessor. Several variants exist nowadays including networked, multi-tasking, concurrent and 16-bit versions of CP/M.

CPU

Central Processing Unit.

Data set

A collection of data items.

Debug

To eliminate errors within a program.

Directive

An assembler operation based on a particular assembler facility (rather than a facility of the processor that code is being assembled for). Sometimes called a *pseudo-operation*.

Editor

See *Text editor*.

File

A set of data items held on diskette, tape or other medium.

Floating Point

A means of representing numbers in the binary equivalent of *scientific notation* by specifying an *exponent* and a *mantissa*.

Hard copy

A printed listing of some computer output as opposed to the output displayed on a VDU screen.

Hashing

or *Key to Address* transformation is a collective term used to describe the techniques for calculating the address of a data item (or data item set) by using a mathematical function of the search key.

Hexadecimal

A base 16 numbering system using the digits 0–9 and the letters A–F.

Iconic

A picture representation using *Icons*.

I/O

Input/Output.

Isomorphic

Systems of similar structure.

Interrupt

An externally instigated request which, if accepted, causes the processor to save its current status and perform some required function. When the function has been completed the status of the processor is restored and control handed back to the interrupted program.

Label

Rectangular shaped paper, often sticky, used for placing identification markings on objects.

Label

An identification name used in an assembly language source code to refer to a particular section of coding.

Linked List

A set of data items linked together by using *pointers*. Sometimes called *chains*.

Memory map

A diagram showing the allocation of the various parts of memory chosen for a particular system or program.

Mnemonic

Pronounced *nem-mon-ik* – an aid to memory. Assembly language mnemonics are designed to help you remember what functions the various instructions perform.

Object code

Machine readable code which has been produced by translating the mnemonic form of an assembly language program into binary code.

Octal

A base 8 numbering system.

Operand

An 8-bit or 16-bit value upon which an instruction will operate.

Operating system

A collection of routines which perform the I/O and other hardware dependent chores that are needed for a computer to function.

Page

256 bytes.

Peripheral

Any external or remote device connected to a computer system, such as a printer.

Pointer

An address, record number or other indicator which specifies the next item of a data set taken in a specified logical order.

Program counter

A 16-bit register used to determine from which location the next instruction should be fetched.

Pseudo-operation

See *Directive*.

Repetition

Repeating a set of actions a given number of times.

RAM

Random Access Memory.

ROM

Read Only Memory.

Sequence
Operations following each other in time.

Set
A collection of items.

Software
Any program or routine for a computer.

Source code
Text version of an assembly language program. Assembly of such source code will produce either the object code form directly or a hex form which is translated into object code form by using a *loader* program.

Sort
To arrange a data set in a specified order.

Stack
An area of memory reserved for storing data in a Last In First Out (LIFO) basis.

Syntax
The formal structure of a language.

Text Editor
A program which enables text to be written, manipulated, stored and so on. Word processors are sophisticated text editors.

Tree
A type of data structure.

Two's complement
A numerical representation in which positive numbers are represented as ordinary signed binary but negative numbers are represented by complementing the number and adding one.

VDU
Visual display unit.

Warnier diagram
A design diagram which uses sets of hierachical curly brackets to indicate the logical structure of a problem, program or system.

Zero page
The first page of memory, addresses 0000_{hex} to $00FF_{hex}$.

INDEX

16-bit shifts – 89
16-bit transfers – 32
8-bit transfers – 31
Absolute addressing – 61
Accumulator – 27, 31, 32
ADC – 33, 85
ADD – 33, 84
Add with carry – 85
Addition – 84
Addressing – 59
Alternation – 47
Alternation, 'binary' – 47
Alternation, complex – 56
Alternation, simple – 56
AND – 33, 69
Architectures – 19
Arithmetic shift left – 70
ASCII – 13, 15, 21, 93
Assembler – 21
Assembly language – 19
Assembly language tree structures – 134
Base address – 62
BASIC – 14, 22, 38, 127
BCD – 83
Binary coded decimal – 83
Binary multiplication – 88
Binary numbers – 20
Binary trees – 122
BIT – 33
Bit addressing – 64
Bit map – 126
Block transfer instructions – 32
Boolean Algebra – 158
Bruner, J. – 3
Bubble Sort – 119
Buffer – 99
Byte – 27
CALL – 33, 42, 51
Carry flag – 50, 51, 85, 87
Circular lists – 98
Comments field – 22
Computed addressing – 62
Conditional relative branching – 55
Conditional relative jump – 42
Conditional subroutine call – 51
Conditional test instructions – 33
Connect Four – 64
Control character – 13, 15, 48
CP/M – 22, 24, 43, 147, 148
CPDR – 114
CPIR – 114
CPL – 73
Data processing instructions – 33
Data structures – 93
DEC – 114
Descendants – 121
Descriptor – 92, 98, 101, 103, 135
Displacement – 61, 62, 67, 68
DJNZ – 108, 113
Documentation – 139
Enactive – 2, 3
EQU – 23, 41, 111
Exclusive OR – 73
FIFO structures – 98
Flags – 28
Floating point numbers – 82
GOTO – 50
Hexadecimal numbers – 20, 29, 150
Iconic – 2, 3, 4
Immediate addressing – 60
Implied addressing – 59
Indexed addressing – 62
Indirect addressing – 63
Instruction field – 22
Instruction format – 34
Integers – 79
JP – 39
Label field – 22
Labels – 23
LD – 32, 34, 40, 59
LDD – 32
LDDR – 32, 103, 104
LDI – 32
LDIR – 32
Left shift – 90
Library routines – 142
LIFO structure – 95, 137
Linked list – 95, 96
Load instructions – 31
Logical opposite – 10
Machine language – 20
Mask – 69
Microprocessor – 19, 27
Mnemonic – 20
Multiple-byte integers – 81
Multiplicand – 88
Multiplication – 88
Multiplier – 88
Mutually exclusive operations – 9
Node – 121, 128
Null word – 110
Object code – 34, 144
Octal – 20
Op-code – 34, 39, 153
Operand – 59
Operating systems – 24
OR – 33, 72, 114
ORG – 39, 40, 41
ORG directive – 24
Osborne-01 – 148, 151
Overflow – 81
Page zero – 31
Pages of memory – 31
Parent – 121
Parent pointers – 137
Partial products – 88
Pointer – 149
Pointers – 96, 98
POP – 32, 110, 115, 118, 137, 138, 154
Procedure – 8
Program counter – 28, 42
Program layout – 141
Pseudo operation – 23, 41
PUSH – 32, 110, 118, 137, 138, 154
Queues – 98
RAM – 27
Register addressing – 59
Register indirect – 64
Registers – 19, 27
Relation matrix – 126, 134
Relative addressing – 43, 61
REM statements – 22, 147
Repetition – 37, 38
Report generator – 8
RES – 33
RET – 116, 118
Right shift – 91
RLA – 137
ROM – 27
Root – 121
Rotation instructions – 90
RR – 137
SBC – 87
Sequence – 37
Sequential searching – 113
SET – 33
Sets of operations – 10
Short addressing – 61
Signed binary – 80
Simple alternation – 47
SLA – 70
Sort trees – 120
Sorting – 119
Source code – 21, 111, 140, 144
Stack – 28, 95, 137
Stack pointer – 28, 40, 41
Standard mnemonics – 21
Static tables – 94
Status word – 28
String space – 103
SUB – 87
Subroutine – 42
Subtraction – 87
Subtree – 129, 130
Symbolic – 2, 3
Syntax – 22
Syntax errors – 143
Tree Sort – 120
Tree structure – 98
Trees – 98
Two's complement – 61, 80, 81
Unsigned binary – 79
Warnier diagram – 4, 7, 157
Warnier, J. D. – 7, 163
XOR – 33, 73
Z-80 – 5, 20, 27, 28
Zero flag – 42, 100
Zero page addressing – 61

Teach Yourself Assembler: Z80 is for all personal computer owners with a Z80 based micro interested in learning to write assembler programs. Although writing in BASIC can be fun, Paul Overaa shows you how to get even greater satisfaction and pleasure by creating your own programs in assembler.

The author's clear, structured style is particularly suitable for those tackling assembler for the first time. The author shows how the fundamental principles are exactly the same as those applicable to high level languages.

This book was developed following the publication of the author's highly successful Teach Yourself Assembler series in *Personal Computer World* magazine. This book is a much-expanded text, specifically addressed to Z80 programmers. There is a companion volume for the 6502 processor.

£7.95

ISBN 0 7126 0549 5
CENTURY COMMUNICATIONS LTD

BEST OF PCW

TEACH YOURSELF ASSEMBLER Z80

CC

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/

[FRA] Ce document a été préservé numériquement à des fins éducatives et d'études, et non commerciales.
[ENG] This document has been digitally preserved for educational and study purposes, not for commercial purposes.
[ESP] Este documento se ha conservado digitalmente con fines educativos y de estudio, no con fines comerciales.