ADVANCED Z80 MACHINE CODE PROGRAMMING

William Nitschke

"If you already know the fundamentals of programming in Z80 machine code, and now want to expand your skills, to tackle advanced, serious and practical program development, this is the book for you . . ."
Tim Hartnell

Interface Publications
LONDON · MELBOURNE

ADVANCED Z80 MACHINE CODE PROGRAMMING

William Nitschke

ADVANCED Z 80 MACHINE CODE PROGRAMMING

William Nitschke

To Mom and Dad

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

First printing August 1985

ISBN 0 907563 90 2

Books published by Interface Publications are distributed in the UK by WHS Distributors, St. John's House East Street, Leicester LE1 6NE (0533 551196) and in Australia and New Zealand by PITMAN PUBLISHING. Any queries regarding the contents of this volume should be directed by mail to Interface Publications, 9–11 Kensington High Street, London W8 5NP.

Cover design – Richard Kelly
Printed and Bound by Short Run Press Ltd, Exeter

CONTENTS

CHAPTER 5 – THE COMPUTER GAME

CHAPTER 6 – THE BUSINESS PROGRAM

CHAPTER 7 – THE VICIOUS CIRCLE

CHAPTER 8 – SPECIAL APPLICATIONS

APPENDICES

FOREWORD

This book is intended to fill a sorely missing gap in available microcomputer literature. Rather than deal with the elementary concepts of Z80 architecture, this book examines advanced, serious and practical machine language programming. Basic texts deal with the 'spelling' and 'grammar' of Z80, this book is about the 'writing' process.

Generally speaking, it has been designed to cater for three types of programmers; beginners who want to supplement their own basic concepts book, and who want to tackle the more complex aspects of the instruction set in greater detail; novice or intermediate programmers who now want to develop software in a professional manner; and, I also hope, experts who wish to view new methods, programming styles and algorithms, to add to their library of programming knowledge.

The programs selected for the inclusion in this book were done so to encourage both professionalism and commercialism. Whether you are serious or not about marketing what you have written is unimportant, but you must be serious about getting the job done.

The first part of this book reviews the complex aspects of the Z80; flags, miscellaneous instruction code, logical operations and other bit manipulations, with a view to their practical application. Moreover, the first chapter looks at the problems of program writing in detail, from runcharting and flowcharting, to developing a practical working plan. Selecting appropriate working tools such as monitors and editor/assemblers are also discussed. The principles of *modular programming,* a method of coding that reduces software development to a comprehensible level is examined.

Besides discussing programming style in general, the first part of this book also looks at writing smaller and faster programs, and debugging describes the most common programming errors associated with the Z80, popular misconceptions, and then check-lists to help you approach the debugging process in a much more efficient manner.

The third chapter deals with communicating, whether this is displaying a string on the VDU, or reading a decimal or hexadecimal string from the keyboard.

The core of the book looks at popular programming applications. Starting with the basics of organising information, generating arrays and tables, block moving, shifting and erasing, string manipulations, data compression and command tables, it quickly moves on to advanced applications like word and sentence decoding, finishing the chapter with a detailed analysis of an 'adventure game'.

Chapter five is dedicated solely to game programming, and considers random number generating, maze generating both in two and three dimensions, animation, arcade games, and various other essential programming techniques.

Chapter six turns to more 'serious' applications. The complexities of sorting strings and numeric data are taken down to a more understandable level, and bubble sorting, Shell sorting, and Quick sorting algorithms are reviewed. Binary and block searching, and the concepts of database management are also covered. Finally the chapter concludes with an examination of a pseudo-computer language, to help you develop your own high level languages.

Chapter seven is about machine language editing, break-pointing, single-stepping, and disassembling. These various processes are scrutinised in detail.

Chapter eight explains how to generate speech synthesis and sound effects. The complete set of 'undocumented' Z80 instructions, which includes four more general purpose 8-bit registers, and special shifts, are finally documented. The chapter finishes by explaining how to write a program for commercial publication, what is expected from such software, how to document it for submission, and what returns you are expected to receive.

The appendix section of the book includes a glossary of machine language terms, hexadecimal to decimal conversion, flag condition and instruction code summaries, and a *complete* listing of the Z80 instruction set.

I hope the programs included in this book will help to rid yourself of the question 'how do I write?' and let you concentrate on the most important question of all: Now, what do I want to write?

William Nitschke.

CHAPTER 1 – DESIGN

This chapter looks at the concepts behind program writing and the technical difficulties involved. The first and most obvious problem is setting up and organising a large program. Without proper organisation it becomes not only confusing and difficult to update, but larger, and more likely to go wrong. A program must also be moulded around the internal organisation of a computer. Efficient programming tools are needed, like sophisticated editor/assemblers and monitors, so their advantages, drawbacks and requirements are considered. Finally, program debugging is discussed in detail.

1.1 WRITING A PROGRAM

You have to design a program before you can begin writing. That means flowcharting, runcharting, thinking out algorithms, rewriting, revising, planning, and then doing it all over again until you get it right. If you have ever had any experience programming high level languages, then you probably never saw an overwhelming need to write down your thoughts. These were gradually developed as you worked on some other aspect of the program. A few lines of comment covering up anything you would have to fill in later. Get these thoughts out of your mind at once. This is not only poorly amateurish, but frankly, an impossible way to program – at least in machine language.

If you don't really see the need to flowchart, then perhaps machine language is not your language? Better still, take a cold shower. Good. First of all, why do you want to write this program? As a mental exercise? To do a job that cannot be done in a high level language? To do something that can be done so much better in machine language? To sell it and make money?

I have adopted a commercial attitude throughout this book for a number of reasons, besides the obvious monetary potential. It encourages professionalism, and best of all, realism. That is to say, there is nothing wrong with using techniques that are not the very fastest, and most memory conservative, as long as it is easier for the programmer to work with (the realism). Likewise however, sloppy programming and designing is inexcusable (the professionalism). Programming is about compromising between you, the Z80, and very probably, your editor/assembler.

A program can only be as good as the programmer, who is limited by technical know-how and the physical limitations of the computer system. Very little can be

done about system limitations. You already know what you are mentally capable of, so make sure you know your computer inside out. Take into account all of the following factors:

CLOCK SPEED. Computers with a clock speed of 3 MHz or more are capable of almost all currently popular computer applications. Luckily the Z80 is a fast processor and is usually set to run at a high speed. Even half this speed with good programming is capable of commercial software. But high processor operating speed could have its disadvantages. Badly built computers are prone to video display flicker. A section of code may need to run faster or slower than others, particularly for sound and/or animation. Since the computer can only do one thing at a time careful planning is essential.

MEMORY. Popular programs like games, word processors, utilities, languages and business tools normally vary in length, but average out to between four and 20K (unless it's a high level language compiled into machine code). Powerful word processors and other software of this type would reserve text areas of 20-50K. Programs like mini-word processors with only a few K of text area are exceptable when the average system (being sold) has severe memory limitations. The power and versatility of a program often depends on its size. Although larger is not necessarily better, as the better programmer is able to write smaller programs.

GRAPHICS. There may be problems representing information on small video displays, and particularly on systems with less than a 50 column display. Graphics and colour will be a determining factor in the playability of many games and the readability of business information in the form of graphs and tables. It may only be possible to access high resolution graphics by deactivating normal alpha-numerics. Graphics may require complex addressing techniques that will make machine language programming difficult to say the least.

PERIPHERALS. In database management a mass storage device is a critical aspect of programming. Cassette recorders are slow, and allow only sequential data storage. Mini and micro floppy disk drives have limitations according to the brands compatible with the system, the quality of the drive, popular operating systems used, and cost per track. Hard disks may not be available or prohibitively expensive.

SOFTWARE. Your program may have to use 'system' programs or be compatible with a disk operating system. The programmer will also have to consider where it must load in memory to avoid interference with other programs, and the form it takes loading into the computer to be compatible with system program loading utilities.

Once you are sure your system is capable of the task set for it, and you're capable of programming it, it is time to start designing. Plan exactly what you want it to do and how you are approximately going to do it. Note down all the ideas that come into your mind, adding footnotes if necessary that mention suitable machine instructions. Draw boxes and arrows to connect up sections of logic and to isolate others. A lack of programming experience makes this process painfully difficult for the novice, so it is essential to make your first programs small and simple, or at least base it on a simple algorithm.

If you seem bereft of ideas, unable to find a suitable way to approach a problem, don't worry for now. The central purpose of this book, more than any other, is to give you this necessary background information. In a nutshell, to supply you with those 'ideas'.

What is more important, the algorithm or the machine code? If you answered, *the algorithm,* you would be wrong. If you answered the *machine code,* you would be wrong again. In low level language programming each is dependant on the other. Many people advise you to work out the algorithm first and then worry about the coding. By that time it's too late. Machine language puts serious restraints on the programmer. Had you designed an algorithm that required fractions or decimal places you would have real trouble writing it in machine language. Even simple operations like multiplication and division require a fair amount of coding, and are usually avoided by professionals. Your algorithm must be based on the Z80 instruction set, which is why it's so important to have a complete understanding of it.

Blending the algorithm and coding stages together creates real problems. And these strike beginners, novices or occasional programmers the most. Either the machine language program has a bug in it, or worse, the algorithm is wrong. And that is very easy to do in machine language. Logical errors will be discussed further on in this chapter.

You should already be familiar with flowcharting; a logic diagram that documents the algorithms for a program. Triangular shapes hold questions. Squares contain brief descriptions about the routine's purpose. Both are joined together by arrows pointing in one direction, showing the logic 'flow'. On the other hand, a runchart documents the execution 'flow', the movement from one routine to another. A program usually needs only one runchart. Flowcharts are necessary for each algorithm. Diagram 1.0 is a flowchart of a video game, diagram 1.1 is its runchart.

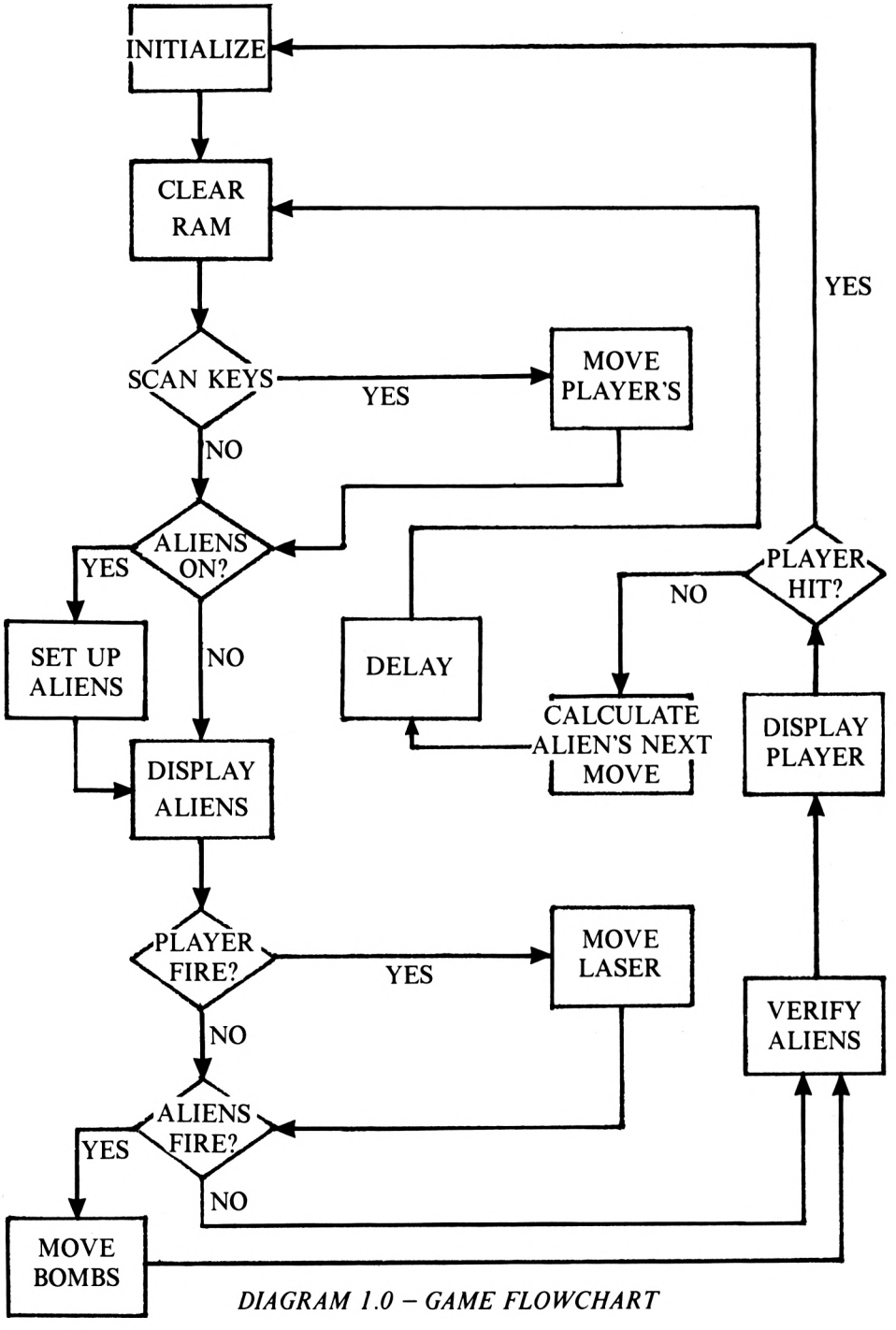


DIAGRAM 1.0 – GAME FLOWCHART

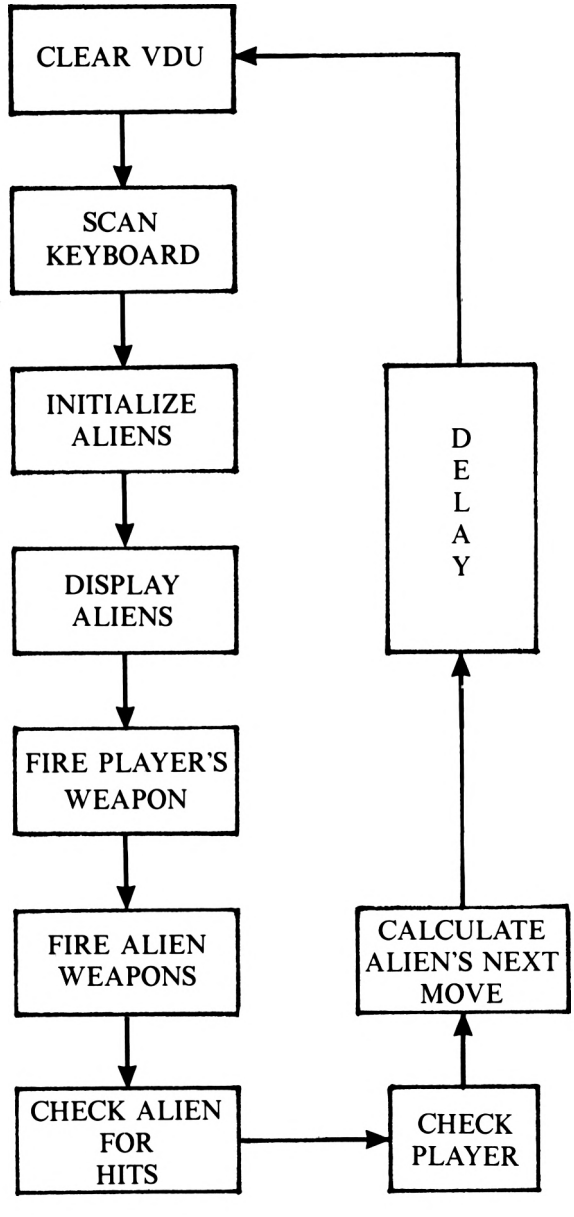


DIAGRAM 1.1 – GAME RUNCHART

Adding detail may not seem important when you start designing a program, since detail can be kept in your head, and later directly coded into your program. But, consider that you may want to later update or improve what you have written, weeks, months or even years later. If you intend to write software for commercial purposes, or possibly design something with that in mind, competition will make the upgrade process essential.

Even when writing for personal use in business or for a hobby you will probably need to introduce new features to expand, enhance (such as taking advantage of a new peripheral) or solve problems that you did not previously have to worry about. Write notes aimed at telling how to update for future modification.

While you can keep the idea of the program in your head during coding, you will eventually need to make corrections and add code you had forgotten in design stages or left out since the problem was more complex than you had first thought. These distractions will get you lost, unless the written code is heavily documented.

Once you have the structural layout of your program, avoid jumping straight into computer coding. Every piece of code, however trivial it seems at the time, should be written down. Buy yourself a large notepad (with pages that can be removed without the rest of it falling apart) and start scribbling with your pen. Once you have finished the initial stages, take a long look at your program, adding anything you have left out, or remove unwanted extras.

Never worry about the 'frills' (headings, copyrights, or strictly aesthetic additions) until after the program is very nearly finished. Make the other relevant corrections, and if necessary, rip it up and start again or use it as a guide for your second attempt. At this stage look for repetitious sections of code. Anything that can be used as a subroutine will save memory and typing. A video display unit has a narrow display enabling you to see only a small portion of code at any one moment (the most 25 lines), while a quick flip of a page can give you a general overview. Moreover, not limited to a computer console, you can add or improve your program wherever you take your notepad.

Sections of your program have to be typed in as you go along, tested and debugged. Develop the human elements first. This includes keyboard communications between the program and the operator, and the visual display of information. Doing this you can begin testing and debugging virtually from the start, and if you are lucky, *see* what is going wrong – which is better than knowing something is wrong, but not what.

Your time should be split up into consecutive periods of writing, programming, and debugging. The best time to start typing is when the code has some sort of basic shape. Depending on the program, testing of the 'skeleton' should begin at least by

the time a quarter of coding has been done (but probably well before). There are important mechanical and psychological reasons for doing this. It allows you to narrow down your bugs before the program gets too large and confusing, and it gives you the confidence to press ahead with the project. After it is working, continue with drafting.

Redrafting is time-consuming, frustrating and inescapable. The problem you are trying to solve, or the task you are trying to perform is probably more difficult than first realised. (If we first realise how difficult the task is we would probably opt for writing something else.) Sometimes you may have to discard your work and start from scratch. Either your code becomes so complicated that it would be faster to begin again, or you have not properly thought out the problem. A *structural error* or *logic fault* eventuates. In other words, a bug so fundamental to the program that it has to be scrapped; or not do what was originally intended. It is an answer to the wrong problem.

Do not put anything off at the design stage. If you think one aspect of your program is complicated or tricky, do it first. It will give you that confidence needed to finish the job. And confidence is virtually essential for success.

Computer programming is by no means an old art, but it is striking to observe that people on their own tend to solve computer problems in a similar manner – given time. What has emerged is *modular programming,* an excellent approach to writing in machine language. Code is subdivided into separate *modules,* each a self-contained portion of logic. Each *module* has further, smaller, *modules*. A program written on this premise is thought of as a series of tiny units, making the writing process more attractive.

Ideal *modules* can be changed or replaced without affecting other parts of the program. It is not the code that counts therefore, but the results they return on exit. Subroutines used by more than one module should be placed in a separate section of memory. These subroutines are called the 'subroutine package' or 'system module' and are rated at the lowest code level.

While modules, in general, can be transferred from one computer to another with only a relocation, system modules are almost inevitably machine specific. However, not too much should be made about this point. Higher level modules (subroutines CALLed before others) could be machine specific as well.

Taking an example, a sophisticated word processor can be broken down into I/O routines, text editing, file handling and printer driver modules. The text editing *module* can be further subdivided into insertion, deletion, overtyping and so on. The insertion *module* into move text, input character, display character, and buffer handler. But 'input character' and 'display character' are routines used in other

parts of the program. These are the system modules. The remaining code makes no further CALLs, and has therefore reached their lowest level. These are self-contained modules.

Modules are potentially reusable and replaceable. When they are used many times they become part of the *programmer's toolkit,* or collection of commonly used subroutines. So a large program should be looked upon as only a collection of small programs, which are themselves only a collection of subroutines. This way, the task is no longer so daunting.

A program has three physical characteristics. The initialiser, where registers and memory are loaded with data when the program first begins. The *modules,* making up the actual machine language. And lastly, the variable table, where constants, variables, buffers and other data are held. Since the initialiser is the first code to be executed, it is often at the beginning, though its actual location is of no consequence. Most programmers put it at the end of their programs, so their editor/assemblers don't repeatedly list it.

I advise you to place the variable table at the end of the program where it has less chance of being accidentally modified by the *modules* and where it is unlikely to overwrite the machine language. In *Modular programming* the Main Body is linked together by a execution control routine that tells it the order of subroutine execution. A typical program would like this:

```
START   DI
        CALL    INIT
LOOP    LD      SP,ØFFFFH
        CALL    SUBR1
        CALL    SUBR2
        CALL    SUBR3
        JR      LOOP
```

```
SUBR1   . . .
        RET

SUBR2   . . .
        RET

SUBR3   . . .
        RET

SYSTEM  . . .
        RET

INIT    . . .
        RET

DATA    DEFB  Ø
```

The program begins by executing the initializer, and then the subroutines SUBR1, SUBR2 and SUBR3. These execute further subroutines, which execute still further subroutines. The low level modules CALL the SYSTEM modules. The advantages are:

- The execution control loop serves as a starting place, or point of reference, for the rest of the program.

- Further subroutines can be placed in the execution control loop as the program is developed or updated.

- The structure encourages the use of subroutines, so saves memory.

- Since the program is divided into separate *modules* it is easier for the programmer to study, update and enhance.

- Bugs are narrowed down to new subroutines as the program grows.

The stack pointer reset instruction, in the execution control loop, continually keeps the stack from doing any damage to the program. It also eliminates bugs caused by accidental stack pointer overflows. Naturally when executed, the stack pointer should be set as far away from the program as possible, and all data PUSHed onto the stack must first be recovered to avoid its loss.

Keeping this in mind, the instruction no longer makes it necessary to RETurn from a subroutine. (You should try to avoid doing this as much as possible, since it defies the purpose of having *modules* in the first place, but in rare circumstances this flexibility comes in handy.) *Modules* have nothing in common with each other, and must therefore use the stack in an organised fashion. Anything PUSHed on, must be POPed off before exiting.

The disable interrupts (DI) instruction is used to cut out possible interference from a disk operating system or other programs.

Since the program always RETurns to the beginning of the execution control loop there are several inherant advantages. Games can have background sounds, business programs, languages and utilities can have control keys that override all routines (without needing to check the key separately in each *module).* By linking the program together using CALL statements the programmer can easily change the order of operation of routines, execute some routines more than others, and make programs look like they're doing many things at once (essential for animation).

Examine a typical initialization routine:

```
INIT    LD      HL,BUFFER
        LD      DE,BUFFER+1
        LD      BC,102
        LD      (HL),0
        LDIR
        LD      A,5
        LD      (ADDR1),A
        LD      BC,9999
        RET
```

Although if many addresses have to be loaded with data it is better to block move it into place rather than register load it:

```
LOAD    LD      HL,STORE
        LD      DE,DATA1
        LD      BC,50
        LDIR
        RET
DATA1   DB      0
DATA2   DB      0
DATA3   DW      0
DATA4   DW      0
        .   .   .
```

```
STORE   DB      34,211
        DW      2211,60121
        .   .   .
```

By placing buffers, byte and word storage locations in one memory location the whole section can be cleared using one block move as shown in INIT. Data to be zeroed (program variables and temporary storage areas) should be organised as such:

```
BUFFER  DS      100
ADDR1   DB      0
ADDR2   DB      0
ADDR3   DW      0
```

Placing reserved bytes at the end of a program enables the programmer to easily add and delete them. The variable table should always be cleared before placing data in it. What might be zero the first time a program is executed, could be any value the second or third time. The result, possibly disaster.

1.2 MEMORY-MAPPING AND PORT ADDRESSING

Machine language programs are not computer interchangeable even if the microprocessor is a Z80 or a Z80 compatible. But there are special exceptions to this rule. You may be familiar with CP/M or MSX and other official or defacto standards, designed specifically for software/hardware compatibility. Yet there is a good chance that your computer is a 'loner' and must have software written specifically for it. Even under the best conditions – operating system compatibility – software interchange may not be possible if your hardware can't read differing data formats. Either way, you will have to tackle the problems of machine language input and output before you can consider developing software.

The *memory map* of a computer is the internal organisation of addressable memory. Most Z80 computers have split their memory into ROM and RAM in a specific way, although some computers have interchangeable memory maps – usually a

ROM cartridge that plugs into an edge card. This is convenient, but not enough to make machine language software completely compatible. The memory map specifies where RAM is located, and consequently where your software must reside. ROM, the operating system, other hardware features, and (probably) a high level language is part of this too. Some areas of RAM are used by ROM as variable data storage and is termed 'reserved RAM'. Remaining sections may be referred to as 'reserved for memory-mapped I/O' or 'video RAM'. These sections all are *memory-mapped I/O* (input/output).

When an I/O device sends or receives data from an address it is termed 'memory-mapped'. Incoming information – the press of a key – a byte from a mass storage device – is loaded into the address as a type of 'data depot'. This process is completely independant of the CPU. Alternatively, a byte can be sent out via an address – displaying information on the video display – sending data to a line printer – using the same method. One address can perform both functions. These addresses are not true memory, although the Z80 is capable of manipulating it (with the exception of storing data permanently) in the normal fashion.

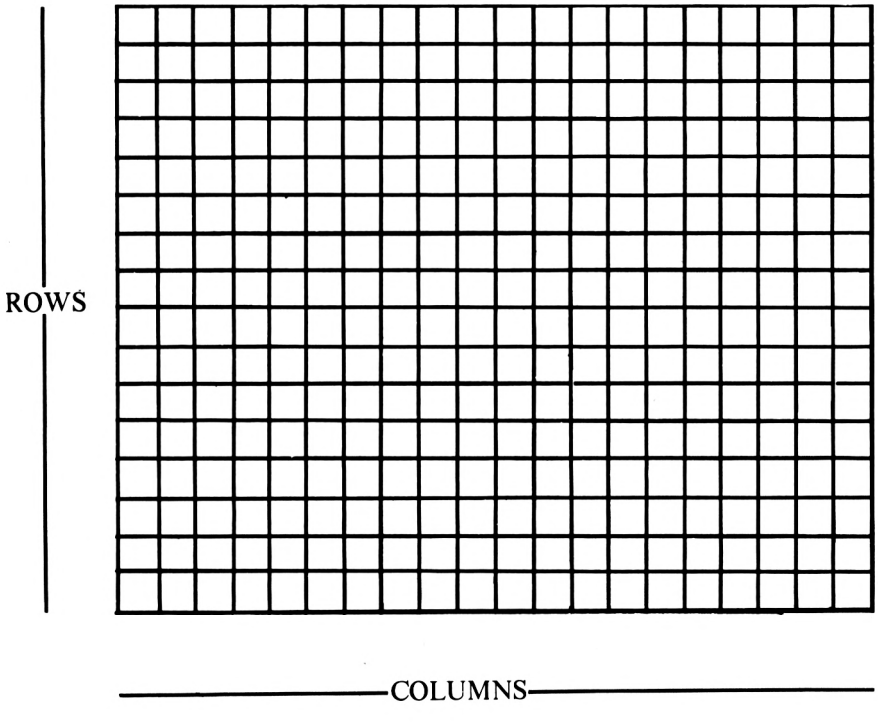


DIAGRAM 1.2 – MEMORY MAPPED VDU

Sometimes timing is important. Since a peripheral cannot operate as quickly as a microprocessor, other information about the status of the device can be placed in the address. For example it would be normal practice to *read* the I/O address of a line printer to determine whether it has displayed the last character, or whether some error (such as out of paper) had occurred. When the condition is no longer true (the bit is no longer sent) data can be loaded into the address once again.

The most common form of memory-mapped I/O is the video display unit (VDU). Some computers use other addressing techniques or a combination, but most are directly addressable. For example, loading address 3C00H with 41H (ASCII character 'A') would display 'A' on the top left-hand corner of the TRS-80 Model I or BBC (both microcomputers) VDU, since this is the first address of their VDU RAM. Diagram 1.2 illustrates how this RAM is commonly portioned out ('mapped') onto your VDU. The intricacies and complexities of addressing VDU RAM is covered further on in this book, particularly in chapter five.

It is easier for the programmer to address memory-mapped VDU I/O, but unfortunately for a number of reasons – mostly cost – other 'compact' and complex methods are used. A high resolution VDU could occupy over 400K of RAM if special addressing techniques were not employed. Data about a character's shape, colour, etc. is therefore often stored in a compressed bit format. Otherwise *bank switching* (putting one block of 64K RAM off line and another on) would have to be used. The fewer RAMs used, the cheaper the computer is to produce.

Port addressed I/O requires you to use the Z80 IN and OUT instructions. Peripherals like line printers or disk drives or cassette recorders could be port addressed or memory-mapped depending on the preferences of the designers. The Z80 can only access 256 ports, so a true VDU cannot be *directly* port addressed. While other I/O techniques exist, such as direct memory access and interrupt driven I/O, they are less commonly used and are only briefly mentioned here.

Much of the assembly language in this book will work on most microcomputers, but out of necessity or preference, I have made some routines machine specific. Rather than include programs that would not work on any computer (merely theoretical programs in other words), I selected a microcomputer of relatively simple design (as far as it concerns us) and a straightforward memory map. It is much easier to learn how to program one computer, and then modify your routines to run on your computer, than to try and fill in the missing code by yourself. The remainder of this section will teach you how to program this computer, and scrutinize and contrast it with other popular models.

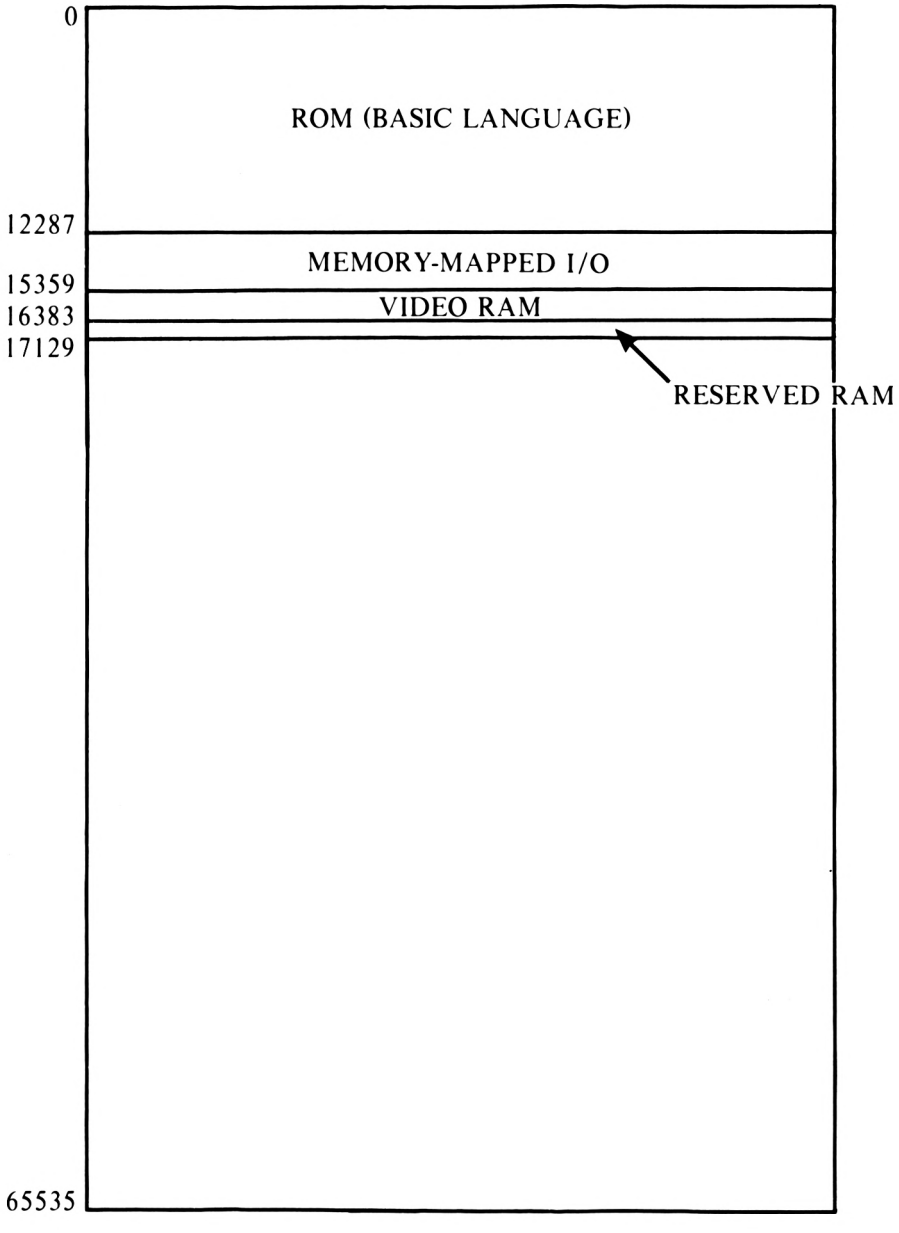


DIAGRAM 1.3 – TRS-80 MEMORY MAP

The computer selected is the TRS-80 Model I, whose memory map is virtually identical to the TRS-80 Model III and IV, which has various other aliases around the world. It will be referred to from now on as just the TRS-80. Diagram 1.3 is its memory map. ROM memory occupies the first addresses. Addresses between 3C00H and 3CFFH, exactly 1K, is mapped directly onto the VDU, which is portioned out into 16 rows of 64 columns. The memory-mapped I/O section is used for keyboard decoding. Everything after 17129 is free RAM for program storage.

Now examine diagram 1.4, the memory map of the LASER-200. In many respects it is similar to the TRS-80, but has a 'reserved ROM' (unused ROM) section and 2K of VDU RAM. This computer has only 32 columns on its VDU, although more RAM is being reserved. Actually, much of this extra RAM is only accessed in 'high resolution' mode, and normal text display occupies less than 1K. Keyboard input is addressed via memory-mapped I/O on both machines. This makes the process of keyboard decoding more complex than port addressing.

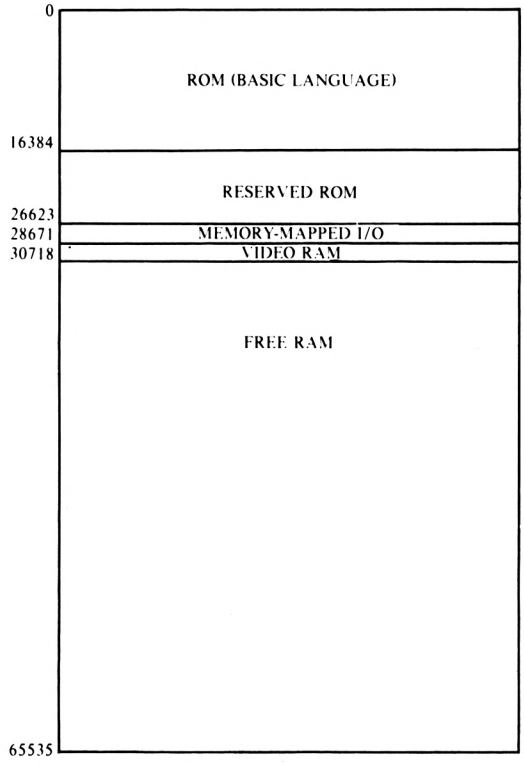


DIAGRAM 1.4 – LASER 200 MEMORY MAP

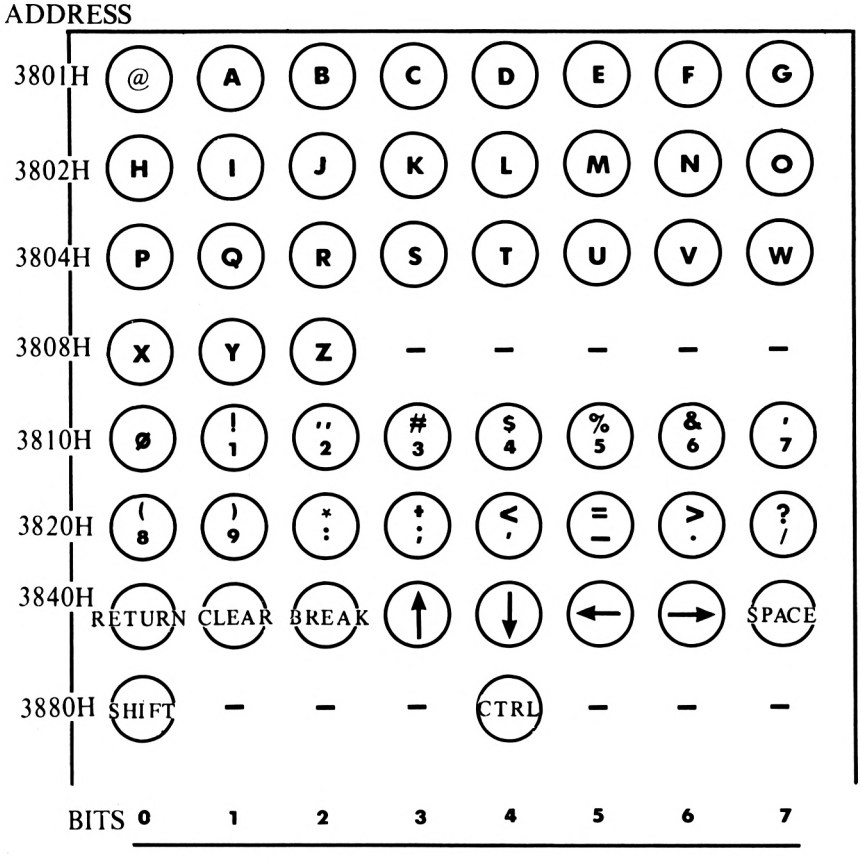


DIAGRAM 1.5 – MEMORY MAPPING OF TRS-80 KEYBOARD

Diagram 1.5 is the 'keyboard matrix' of the TRS-80. When a key is pressed, a bit in one of the addresses 3801H, 3802H, 2804H, 3808H, 3810H, 3820H, 3840H or 3880H is set. It is the job of the programmer to check each bit of every necessary byte. As well, a byte can be used to scan more than eight keys when it is used for more than one purpose, such as upper and lower case, as well as other functions. To scan for the 'A' key an address must be checked:

```
INPUT   LD      A,(3801H)    ;READ KEY
        BIT     1,A          ;IS IT SET?
        JR      NZ,INPUT     ;RESET? AGAIN
```

The keyboard matrix of the LASER-200 is essentially similar, but there are a few subtle differences. Firstly, bits 'go low' (reset) when a key is pressed. And while the TRS-80 matrix is well ordered, the LASER-200's is scrambled. For example, on the TRS-80 addresses 3801H can be used to scan the keys A B C D E F G and 3802H for H I J K L M N O etc. But on the LASER-200 address 6FFBH scans the keys Z X C V B only. Remaining bits are ignored. A routine must therefore be able to decode a keyboard matrix regardless of the characters' order. Listing 1.0 decodes characters between '@-0' on the TRS-80. Modify TABLE to read any keys you want. Order is unimportant.

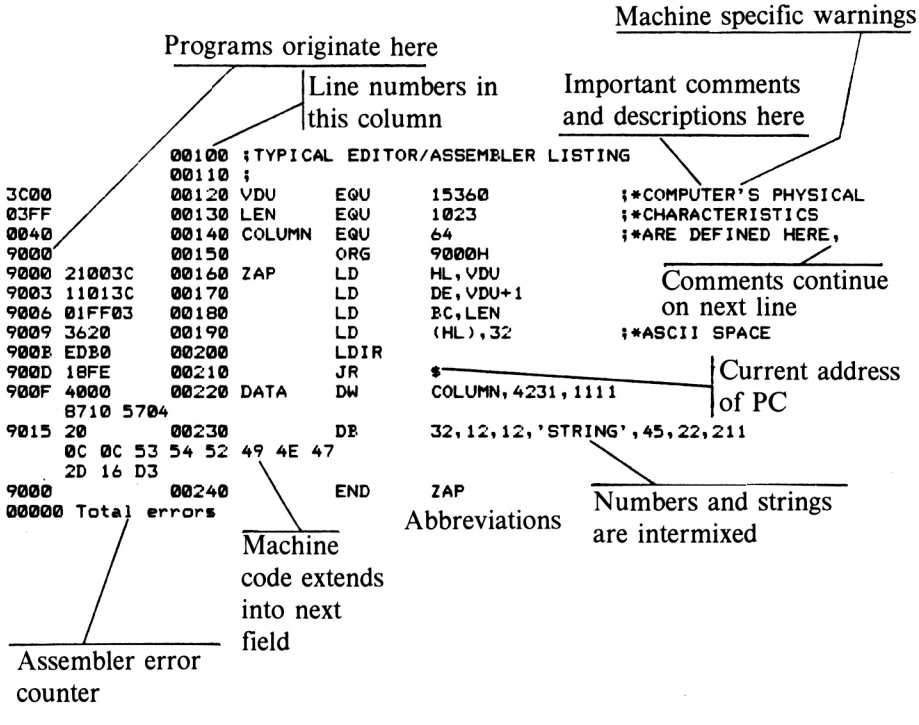
```
                00100 ;MEMORY-MAPPED KEYBOARD DECODING
                00110 ;
9000            00120          ORG     9000H
9000 01FF01     00130 KBDEC    LD      BC,01FFH        ;B=1, C=0FFH
9003 3A0138     00140          LD      A,(3801H)       ;1ST ROW
9006 B7         00150          OR      A               ;ALL CLEAR?
9007 2009       00160          JR      NZ,PRESS
9009 0609       00170          LD      B,9             ;START AT 9TH KEY
900B 3A0238     00180          LD      A,(3802H)       ;2ND ROW
900E B7         00190          OR      A
900F 2001       00200          JR      NZ,PRESS
9011 C9         00210          RET                     ;DONE
9012 0C         00220 PRESS    INC     C               ;INC TABLE POSITION
9013 1F         00230          RRA                     ;SCAN EACH BIT,
9014 30FC       00240          JR      NC,PRESS        ;BY CHECKING CARRY
9016 211E90     00250          LD      HL,TABLE-1      ;START OF CHARACTERS
9019 23         00260 LOOP     INC     HL              ;FIND BYTE
901A 10FD       00270          DJNZ    LOOP
901C 09         00280          ADD     HL,BC           ;ADD C
901D 7E         00290          LD      A,(HL)          ;GET CHARACTER
901E C9         00300          RET                     ;GOT CHARACTER
901F 40         00310 TABLE    DB      '@ABCDEFG'      ;1ST ROW
     41 42 43 44 45 46 47
9027 48         00320          DB      'HIJKLMNO'      ;2ND ROW
     49 4A 4B 4C 4D 4E 4F
9000            00330          END     KBDEC
00000 Total errors
```

The NC conditional jump in line 240 would have to be changed to C to make it read the LASER 200's keyboard, but what if it were not memory mapped at all? A port addressed keyboard can be easier to decode since only one port needs to be read. However, unwanted bits still have to be cleared, and the data may be disorganised. There is no all-purpose routine for a port addressed keyboard. Examine a portion of a keyboard decoding routine for the ZX Spectrum:

```
INPUT   LD      C,254           ;PORT ADDRESS
        IN      A,(C)           ;GET KEY
        CPL                     ;COMPLEMENT
        AND     1F              ;00011111B
        .   .   .
```

Just like the keyboard matrix decoding routine, a similar version must deduce – from the bits in the accumulator – which key has been pressed and get the appropriate character out of a table.

To be able to program a computer you must know the addresses of the keyboard matrix, or the valid port, and the format and addresses of the VDU. If you don't you will have to find them out. Supplementary information regarding some popular computers is included in appendix B of this book.



1.1 EDITOR/ASSEMBLER LISTINGS

Other useful information includes ROM CALLs. These are *modules* used by your computer's operating system software or interpreter. All the computers mentioned so far have useful ROM I/O routines than can be accessed in a single CALL if you know the address. CALLing 2BH on the TRS-80 will RETurn with the correct character in the accumulator:

```
INPUT   CALL    2BH             ;ROM CALL
        CP      'A'             ;CHARACTER A?
        JR      NZ,INPUT
```

Special CALLs such as these may inadvertently change the contents of other registers. The CALL 2BH for example, changes the contents of DE before RETurning. So thoroughly test any ROM CALL before using it extensively. There are times when it's better to use your own routine if you need to decode the keyboard in a special way, or don't want added extras like key debounce.

The label 'GKEY' is used throughout this book to indicate that a ROM CALL or your own keyboard decoding routine is being CALLed. The routines used in the listings in this book always expect the accumulator to RETurn with the appropriate character.

1.3 CHOOSING AN EDITOR/ASSEMBLER

An editor/assembler program converts Zilog's symbolic mnemonics into machine readable numeric code, and is used as a flexible method of editing 'source code' or assembly language. The following is a summary of popular capabilities found in many editor/assemblers, few feature all of them:

- compiles Z80 and 8080 chip mnemonics
- compiles undocumented Z80 instructions
- file handling
- screen text editing
- macro processing
- automatically defined macro processing
- cross referencing
- special assembler directives such as conditional assembly
- direct memory assembly
- assembly from disk file
- instant assembling.

Of course there are many other features such as built-in hex/decimal/binary calculators, unlimited nesting, saving text without line numbers, menu options and so forth, but the major features are in the list. Let's examine the values of each in turn.

Unless you work with an 8080 microprocessor, the first option is of limited value. Worse still, the editor/assembler will probably be larger – leaving you with less text buffer – than just a conventional Z80 assembler. The 8080 chip has been completely superseded (or close enough) by the Z80.

Compiling undocumented Z80 instructions (refer to chapter eight) is beneficial for your own personal use, but I recommend you avoid using them, even if the editor/assembler offers them, on the off chance that Zilog will modify the Z80 chip.

If you are serious about programming, a disk drive is essential. Even more essential is a disk based editor/assembler that can generate relocatable files, not just save and load the text buffer. A file handling editor/assembler should be able to read a subroutine from disk and insert it in your program for assembly, without ever having the assembly language in the text buffer. For example the following could be a valid command (the actual syntax used is unimportant and will vary with each editor/assembler):

```
        ORG     9000H
START   LD      A,20H           ;ASSEMBLY,
        LD      (15360),A       ;LANGUAGE,
        CALL    33H             ;PROGRAM,
        CALL    2BH             ;HERE
        #GET    DELAY           ;READ FILE
        JR      START
        END     START
```

The ' #GET DELAY' will search the disk for the file 'DELAY' and insert the assembled text between the code in program 'START'. Using this feature text between the code in program 'START'. Using this feature you can build an assembly language 'toolkit' – a collection of useful subroutines that can be merged into each job. If the assembly language 'DELAY' were:

```
START   LD      BC,5000
DELAY   DEC     BC
        LD      A,B
        OR      C
        JR      NZ,DELAY
```

The assembler should be able to avoid a multiply defined label conflict. Conflict between your program and toolkit labels should never occur. Unless you want to assemble and merge in memory a number of separate *modules,* you will have to assemble files together by reading a disk. Even the smallest most compact editor/assembler text buffer can hold only a small portion of a machine language program. An average is about 4K of machine language (roughly 32K of assembly language).

Full screen text editing is a quick and convenient way of modifying source code as opposed to line editing. In line editing the operator types in a command like 'E50'. Line 50 is displayed on the VDU and the operator can then delete, change, hack, extend, insert, find etc. by moving the cursor over the desired character (by pressing the spacebar or backspace) and keying in the particular command. In screen editing the operator moves the arrow keys (or whatever appropriate keys) up, down, left or right, until over the desired line, and then automatically inserts and deletes data by typing over the old material. Screen editing is a more powerful way of editing text, and that is probably what you'll be doing the most of.

A macro is a predefined function or set of instructions that can be repeated many times in a program using a single instruction. In most cases it is better to use a CALL instruction. While invisible in assembly language, a macro generates the complete set of instruction code it is designed to represent. Yet there are times when the instruction sequence is too short to justify a separate CALL (three bytes in length) or it is just not possible to use a CALL. Examine a typical macro definition:

```
SAVE MACRO    !DATA,!ADDR
        LD    A,!DATA      ;DEFINE MACRO,
        LD    (!ADDR),A    ;FUNCTION
        ENDM               ;END OF MACRO
```

The instruction:

```
        LD    A,DATA
        LD    (ADDR),A
```

can now be replaced with a:

```
        SAVE  DATA,ADDR
```

instruction (again, the actual syntax of macro definition and use is unimportant and will vary). Had the two instructions been used ten or twenty times in a program, such a macro would be justified. It can also be used to define instructions otherwise impossible in a CALL instruction:

```
PUSH   HL
PUSH   IX
POP    DE
PUSH   IY
PUSH   BC      and other stack operations.
```

Some macro operations are part of the editor/assembler, so that:

```
        MACROP
```

would:

```
PUSH   HL
PUSH   DE
PUSH   BC
PUSH   IX
PUSH   IY
PUSH   AF      and so forth.
```

A cross reference utility should give you a read out (or preferably a print out) of the memory location of all labels used by your program in alphabetical order with the line number of definition, the value of definition, and all lines using it. This is valuable when you are trying to edit a machine language version of your program.

Special assembler directives allow you to automatically abort assembly on error, or other conditions, or, to list only certain sections of your program during assembly. Different assembler directives avoid conflicting labels, etc.

Direct memory assembly allows the operator to assemble and execute the program while still under editor/assembler control. Usually the area of assembly must be defined beforehand and it is not possible to assemble over the editor (or assembler naturally) or your own text. Nevertheless you should always save your file before a direct memory assembly. Anything can go wrong and it usually does.

Sometimes it is possible to assemble a program stored on a disk file. This is strictly a time saving feature, since you don't have to load the text into the editor prior to assembly. However, if it's possible to assemble and link other files together, the program really has powerful file handling capabilities.

Instant assembling varies from automatic syntax check on line entry to genuine instant assembly. A conventional editor/assembler stores source code in ASCII format, so the instruction 'CPL' would actually take three bytes for each letter and another three to five for the line number. Adding to that the editor/assembler size – at least around 8K – only very small programs can be assembled at once. An instant assembler assembles each instruction as it's input. The 'CPL' would therefore occupy only one byte, plus the three to five for the line number. This increases the text buffer capacity by over three times, and complete assembly is done in a fraction of the normal waiting period.

To redisplay the 'CPL' the instruction must be reconverted into ASCII (disassembled), so such an editor/assembler tends to be at least around 12K in length. Naturally the obvious compactness of the assembly language justifies the deficient 4K.

Many more features exist, and more will appear as new programmers write them and more fresh ideas find a place inside these programs. Hopefully you now know what you should expect from an editor/assembler. If you already have one, does it suit your requirements?

While the only effective way of saving buffer space on an editor/assembler is macro definition and using brief labels, there does exist another, less common program called a monitor/assembler, which virtually does away with the text area problem. At a cost. It behaves like a normal debugging program but also allows the operator to type mnemonic code directly into memory. Labels are disallowed but it does calculate relative jumps and other tedious chores. If such a program also features a single-stepper (refer to chapter eight) it becomes an invaluable way of mastering every aspect of Z80 programming. For difficult programming environments: non-disk based, low memory, then it is the most powerful and appropriate.

As already mentioned, assembler formats vary with every author's preference, but there are many commonly accepted procedures. The standard abbreviations are used in the book:

```
SPACE   DB      40,211,'HELLO',0FFH,'A'
```

and string/numeric data mixing is allowed using the Define Byte pseudo-operation:

```
STRING DB       'MODEL I'
```

as well as Define Message (DEFM or DM). In some editor/assemblers the latter data might have to be rewritten as:

```
SPACE   DEFB    40
        DEFB    211
        DEFM    'HELLO'
        DEFB    0FFH
        DEFM    'A'
```

Furthermore, the superfluous ':' after each label is disregarded. Your editor/assembler may require it. Special assembly features are avoided in the listings in this book. Listing 1.1 (refer to page 19) shows the typical format of assembly language programs found in later chapters.

These 'direct' assembly listings, as printed by a microcomputer, are only used to display parts of complete programs or completely self-contained subroutines. The * symbol denotes that the instruction at that line is machine specific and must be changed to suit your computer. If you don't have conventional directly addressable VDU RAM, further modification for direct VDU addressing is required and may not be denoted by a * symbol.

1.4 DEBUGGING

Debugging is the most frustrating part of programming, so this section will deal with errors found in microprocessor programming, and those peculiar to the Z80 before we begin actual coding.

Programming errors or 'bugs' can be split into three types; logical faults, mis-programming and bad syntax. This section will deal with each trouble spot in turn.

Bad syntax, usually in the form of a typing error, is not necessarily easy to find. While mixing up mnemonics (for example; RRAC instead of RRCA) and similar mistyping causes blatant errors picked up by most assembler programs, many more can pass through an assembly undetected. This could be a 'CPI' instead of a 'CPL' or a 'bad label' – one that is not defined in the program. An editor/assembler may give a default (probably zero) to any label it cannot identify.

Entering wrong values, using incorrect registers, mistyped flags, etc. are difficult to locate errors. When searching for bugs, look for bad syntax first. Examine the following piece of code:

```
        LD      BC,50

LOOP    DJNZ    LOOP
```

The routine, a delay loop of 50, has accidentally loaded C with 50 and B with zero. The actual delay is 256, so if B had been used to repeat a set of important instructions, the bug would have been fatal. Depending on the mnemonics before or after it, such a simple error can easily be missed. A fair proportion of all errors in principle are like this one. Always look for syntax errors first.

Make sure you place a '0' in front of hexadecimal numbers starting with a letter; C3H is invalid and an editor/assembler may accidentally interpret it as a label. Make sure that you have added the 'H' appendage to hexadecimal numbers and 'B' to binary ones. Other common errors include the order of operation in LD, ADD, and SBC instructions; such as LD A,B instead of LD B,A. The correct increment and decrement commands, such as INC H instead of INC HL. Whether the flag is a Z or NZ, C or NC and so on. Is the instruction an LDIR or LDDR? Have values been POPed and PUSHed off the stack in the correct sequence? Should an AND or OR instruction be used? Examine the validity of immediate data, checking for the correct base (decimal, hexadecimal, binary, etc.) and whether instructions have been typed in twice or skipped.

Misprogramming is simply a lack of familiarity with the microprocessor. The programmer is usually unaware of some aspect of the Z80, or 'side effect' of an instruction (ignoring implicit results) or expecting something (for example; the setting of a flag) that is not valid. It may also be a more direct misinterpretation, such as confusion over memory addressing of the stack. These mistakes could simply be bad syntax, but it is also possible to pass over many of them without thought. The sophisticated nature of this microprocessor makes such errors difficult to avoid without a lot of practice.

Misprogramming can be broken down into categories:

- Confusion over number storage in memory. Especially confusing 8-bit and 16-bit number formats.

- Expecting non-existent implicit effects. The programmer might believe that certain flags and registers are affected by specific operations, but are not.

- Not expecting implicit effects. Particular instructions might affect flags, registers and other data without the programmer's knowledge.

Stack operations are a common source of tribulation. Examine the following disaster:

```
        LD      HL,5000
        PUSH    HL              ;SAVE '5000'
        CALL    SUBR1
CONT    NOP
        . . .                   ;CONTINUES HERE


SUBR1   LD      BC,64
        POP     HL              ;RECOVER '5000'
        ADD     HL,BC
        PUSH    HL
        RET
```

These instructions will cause chaos if the programmer wants to RETurn to CONT. At first glance it seems the value being POPed off the stack in SUBR1 is 5000, but the more experienced programmer will see that this is not true. The value recovered in HL is the return address of the subroutine CALL – that of CONT. Adding 64 to HL is really adding 64 to the RETurn address. To retrieve the 5000, the stack must be POPed twice, or preferably avoid PUSHing HL at all.

```
        LD      HL,5000
        CALL    SUBR1
        LD      (VALUE2),BC
        POP     BC              ;2 BYTES OFF
        RET
        PUSH    HL
CONT    NOP
        .   .   .               ;CONTINUES HERE


SUBR1   LD      BC,55
        ADD     HL,BC
        RET
```

Remember for each and every PUSH there must be a POP before any RETurn instruction.

```
        PUSH    HL
        PUSH    DE              ;4 BYTES ON
        CALL    SUBR2
```

```
        POP     DE
        POP     HL              ;4 BYTES OFF
CONT2   NOP
        .   .   .
SUBR2   LD      DE,55
        ADD     HL,DE
        LD      (VALUE1),HL
        PUSH    BC              ;PUSH 2 BYTES
        INC     BC
```

Keep your PUSH and POP operations in or out of your subroutine.

It is also easy to get confused over memory addressing and data. Remember the following rules: IF a register pair is loaded with immediate data, the left most register holds the most significant byte and the right most holds the least significant byte. For example:

```
        LD      HL,4576H
```

will load H with 45H and L with 76H.

If a register pair is used to store a 16-bit value in memory, the least significant byte is stored first, and the most significant byte last. For example:

```
        LD      (5000H),HL
```

will store register H in 5001H and register L in 5000H. Naturally:

```
        LD      HL,(5000H)
```

will load H with 5001H and L with 5000H.

Addresses are not the only bytes stored in reverse order when loaded in memory. The Z80 reverses all 16-bit manipulations, including instruction code. Although for readability the instruction:

```
        LD      HL,(5043H)
```

is displayed with address 5043H, it is actually stored in memory as 43H, 50H with the least significant byte first. This also applies to the addresses of CALL and JP instructions. Number formats in memory are discussed further in chapter seven.

The programmer does not normally worry about reverse order unless editing machine code – since the editor/assembler outputs this format automatically – but it does become important when it is necessary to access half a 16-bit word. The following program loads a single register with the most significant byte of a register pair:

```
LD      (5000H),HL

LD      A,(5001H)
```

or

```
LD      A,H
```

With practice the programmer will become familiar with all implicit effects. Until then, there is likely to be some confusion. Take the instruction:

```
SUB     B
```

It subtracts B from the accumulator without indicating it is one of the operands. An implicit effect of the LDIR instruction is to load the value addressed by HL into DE, increment HL and DE, decrement BC, then repeat the instruction until BC equals zero. Instructions can do a lot of things without you knowing about it.

Be careful that you have the right number in BC when Block moving and Block comparing. Programmers often miscalculate by a byte. You should come to expect some effects automatically, which is good. It is handy to know that a DEC A will set the zero flag if the accumulators new contents are zero. It is bad to expect the flags to be affected by decrementing a register pair. For the correct procedures dealing with implicit instructions, particularly flags, refer to chapter two. This section covers common misconceptions.

The SBC instruction subtracts a register pair *and* the carry flag from HL. If the programmer doesn't wish to subtract carry from the result (as in most cases) it must be cleared using an AND or OR instruction:

```
OR      A

SBC     HL,BC
```

will clear the carry flag (the value of A is not affected) and the exact value in BC will be removed from HL.

The more commonly used instructions with implicit effects include: DJNZ, CPL, NEG, AND, XOR, OR, LDIR, LDDR, LDD, LDI, CPD, CPDR, CPIR, CPI, and SUB.

Logical faults are sometimes more difficult to locate since all executed instructions are valid. Possibly the algorithm is faulty. In which case think through your logic again and look for loopholes, or check that your coding is really following the principles of your algorithm. Experienced programmers can rely on old algorithms – by modifying them to suit new routines – to get most jobs done. Limited practical experience puts you at a disadvantage. Confirm that what you think is happening, is really happening.

Logic errors are also caused by poor organisation of the program. Failing to update a register, accidentally reloading an old value, failing to skip over certain routines or missing important routines – all are attributes of poor program design. Go through the following check-list when searching for logical faults:

- Does the algorithm really work? Can you write a rough version of it in a high level language and test it there? Can you work through it using a calculator and come up with the expected results?

- Does your coding really do everything required of it by the algorithm? Are any routines left out?

- Are all registers loaded with valid data before entering the routine? If a register should be cleared, then clear it in the routine, not somewhere else. Is there a chance that a 'bad' parameter (something impossible or ridiculous) could enter a subroutine?

- Are important values lost during the execution of any part of the routine? Check to see that CALLs, or other sections of code do not change important registers.

- Are all results saved away? Or are the correct registers loaded with correct results when entering a routine?

- Are you updating all counters and pointers? Do you need to increment or decrement any registers in loops?

Another part of the program could indirectly cause the error, but it is reasonable to start searching the routine handling the manipulations going wrong. Write down on a piece of paper all the registers being used and then follow each operation in you mind, writing down the values in the registers as a memory jogger. This, 'play

processor' is an important step in debugging and shouldn't be taken lightly. Try not to skip through this procedure, as you would probably skip a few bugs as well.

Try and look at your code as objectively as possible, rechecking even what you think could not possibly go wrong. If there is a logic fault the bug will probably be picked up by doing this. Preliminary checking should pick out a few errors, although this would seldom be more than half. Load and execute the program, noting carefully what went wrong and where. At times you will have to revert to inserting pauses in your code to locate the flawed sections.

If all this fails, your next step is to check the algorithm. Should the algorithm seem valid, review the machine code again. A fresh look at the code in the morning might help. Try sleeping on the problem. You may even wake up with the answer, or at least a few leads. Expect to spend a fair proportion of your time debugging – everyone else does.

Then are those bugs that no check-list will be able to help you find. In a video game I once wrote, the alien bombs carried through, not only attacking the defenceless Martian city, but my machine code located directly below it. The moral of the story, never rule out any possibility.

If you have given up on a bug, very few alternatives are left open to you. You could try to move the routine to another location in memory, or rewrite it from scratch. Desperate measures! A monitor-debugger program is able to find many bugs that would otherwise cause intense frustration. Try to buy one, or write one.

In summary, here is a final check-list:

1. Check for syntax error or typos.
2. Check immediate data, jump addresses, and memory used for storage.
3. Is all data used in the routine valid?
4. Make sure all expected implicit effects are valid.
5. Make sure you are aware of all implicit effects.
6. Examine the stack for possible errors.
7. See if values are being loaded or saved correctly.
8. Check that all addresses and pointers are being updated.
9. See that register contents are not being accidentally changed.
10. Are your calculations one byte off?
11. Determine if anything is executed that should not be.
12. Make sure registers are not being reloaded with old data.
13. Does the routine have all the code to do everything you think it does?
14. Carefully review the algorithm.
15. See if the routine is being CALLed correctly or being affected by other parts of the program. Is the program properly organised?

CHAPTER 2
– STRUCTURE

There are two critical errors that the beginner or novice usually makes. The programmer lacks familiarity with the instruction set's strengths and weaknesses, and therefore selects unsuitable instructions and less than desirable registers. The only concrete way to solve this problem is to practice, but an overview of the instruction set and individual instructions help. This chapter does that.

Secondly, the programmer ignores 'peculiar' or 'mysterious' instructions, like bit manipulations and many flag operations. This leaves a serious gap of knowledge. A programmer has to feel comfortable with the language to be confident. And without confidence, there is little chance of ever finishing a project of reasonable complexity. This chapter explains the commonly misunderstood.

2.1 CHOOSING THE RIGHT INSTRUCTION

You are probably aware that the Z80 instruction set has very obvious register limitations. Selecting the correct register is often confusing for the novice, so it is important to understand the functions and limitations of each group of instructions. While this section is elementary for the expert, it should be read to refamiliarize the programmer with some of the subtleties of Z80 programming.

There are many ways a routine can be written. The following code clears the VDU by loading the TRS-80 memory map, 15360-16383, with ASCII code 32, a blank space:

```
        LD      IX,15360    ;FIRST POS
        LD      DE,-1       ;OR ØFFFFH
        LD      HL,1Ø23     ;COUNTER
LOOP    LD      (IX),32     ;CLEAR
        INC     IX
        ADD     HL,DE       ;SUBTRACT 1
        JR      C,LOOP      ;OVERFLOW
        RET
```

or

```
        LD      HL,15360    ;FIRST POS
        LD      BC,1023     ;COUNTER
LOOP    LD      (HL),32
        INC     HL
        DEC     BC
        LD      A,B
        OR      C
        JR      NZ,LOOP     ;LOOP IF BC NZ
        RET
```

or

```
        LD      HL,15360    ;FIRST POS
        LD      DE,15361    ;DESTINATION
        LD      BC,1023     ;COUNTER
        LD      (HL),32
        LDIR                ;BLOCK MOVE
        RET
```

or . . .

```
        LD      HL,15360
LOOP    LD      A,H
        CP      64              ;END OF VDU RAM
        RET     Z
        LD      (HL),32
        INC     HL
        JR      LOOP
```

All four routines demonstrate important aspects of machine language programming. The first requires an understanding of flags, the second of logical operations, the third of block move technique, and the fourth – just careful observation. Even more importantly they show that there are many ways to do the same thing, even for a small routine, and the best is often debatable.

The block move routine is faster, but the last one uses less memory and fewer registers. Even so, the block move uses less source code in an editor/assembler, something that is always memory critical. Of course, if the accumulator must be saved, the first routine has obvious advantages. If we were to choose the most readable routine, that used the least registers, then the second one would be the best choice. You probably consider one of these routines as the best, yet in reality, not one is outstanding enough to justify use in every circumstance. But there are *better* ways – it all depends on what has to be done – and what we have to work with.

Before continuing, I would like to emphasize that this chapter is to help you to work with Z80 more efficiently, but the most important thing is to get the job done and not worry about possible logical operations, flags, or unfamiliar instructions that could have been utilized to produce a smaller or faster program. Some methods are better than others, but you should never waste time looking for the very best way. *There is always a better way.* If the routine is going to be large and slow, by all means try another approach. If you choose a two byte instruction fifty times in a program, when there might be a one byte instruction to do the same job, by all means try and find it.

The Z80 has an impressive collection of registers which, while making programming slightly more difficult to learn, rewards us with a very powerful microprocessor. To gain the most out of coding you should be familiar with the

instructions applicable to each register. One of the most important factors affecting the size and speed of a program is register selection.

The register pair AF is made up of the accumulator and flags register. The flags register cannot be manipulated directly, except for CCF and SCF, and its value is dependent upon the results of other operations. The 'accumulator' or A register (the only register with an alternate name) is central to almost all 8-bit operations. If two operands are not specified in an implied instruction, like SUB B, it is usually safe to presume that the other operand is the accumulator. (There are a few exceptions to this rule in the case of highly specialized instructions like DJNZ.) The A register *must* be used to perform logical operations (like AND, OR, XOR), addition (like ADD A), subtraction (such as SUB), compares (like CP), complements or negations (CPL and NEG) and one byte length rotate instructions. It is also the only register that can be used to load memory or store data using direct addressing as in LD (5000),A. Since it is the most important single register, it is used in most routines.

The most important 16-bit register pair is HL. Compared with other register pairs it has about twice as many instructions applicable to it. In particular, it can jump to memory locations using JP (HL) and immediate data can be loaded into the contents of its current address; LD (HL),55, or just incremented or decremented; INC (HL). It is possible to swap its contents with the top of the stack using EX (SP),HL.

The second most important register pair or, secondary register pair, is DE. Its contents can be swapped with HL using the EX DE, HL instruction. By swapping back and forth, it is possible to carry out all manipulations relating to HL using DE. Otherwise it has a reasonable, but limited, instruction subset. It includes; ADC HL, ADD HL, DEC, INC, LD, PUSH, POP and SBC HL.

Register pair BC is the standard counter for loops. Register B is used in the loop instruction DJNZ, and BC in block move (such as LDIR) and block compares (such as CPDR). The C register can be used in I/O instructions such as IN A,(C) and block input instructions. Otherwise it is a fairly general register pair performing all the operations of DE, except for EX DE, HL.

The IX and IY registers are often unjustifiably avoided, but they provide powerful indexing capabilities. While AF, BC, DE and HL can be split up into registers A, flags, B, C, D, E, H and L respectively, index registers are usually not subdivided since these instructions are not documented in Zilog technical specifications. (For special notes on their application refer to chapter eight.) The index registers are commonly used in array manipulations. It must be loaded with a 'base address' or starting address, after which it is possible to directly manipulate values between 0-255 further on in memory.

While the editor/assembler program will accept the LD (IX),A instruction, this is really LD (IX + 0),A. The zero, in this case, is called the 'offset' or 'displacement'. It is possible to ADD IX, LD IX, PUSH IX, POP IX, EX (SP),IX and IY. All registers and direct data can be loaded into the contents of an address pointed to by an index register. While index registers use extra memory – a byte identifier, and an offset byte whether necessary or not, it has special instructions that can only otherwise be found using HL. The following is an example of an offset calculation using HL:

```
STORE   LD      HL,5000H    ;BASE ADDRESS
        LD      DE,25       ;OFFSET
        ADD     HL,DE
        LD      (HL),A
```

Which could be achieved using the index registers by:

```
STORE   LD      IX,5000H    ;BASE ADDRESS
        LD      (IX+25),A   ;BASE + 25
```

The index registers are also very useful when specific locations have to be loaded with immediate data. Using HL:

```
LOAD    LD      HL,(MEM)
        LD      (HL),A
        INC     HL
        INC     HL
        LD      (HL),C
        INC     HL
        LD      (HL),B
```

Using IY:

```
LOAD    LD      IY,(MEM)
        LD      (IY),A
        LD      (IY+2),C
        LD      (IY+3),B
```

And there is the added advantage that the base address in IY is not changed. If one of the other registers are occupied, IX and IY are always reliable alternatives. While they have obvious instruction limitations, it is possible to swap them with HL by using the instructions:

```
        PUSH    HL
        EX      (SP),IX
        POP     HL
```

The instructions JP (IX) and JP (IY) are valid, and register contents plus offset can be directly incremented or decremented; INC (IY + 31). There is no physical difference between IX and IY and both registers carry out the same functions. They are powerful registers that should not be ignored for the sake of a few bytes.

The SP register or stack pointer is not a register pair, but a complete 16-bit register. Most programmers make the mistake of studying the conventional registers but ignore the specialised ones. This register, like the Flags register, is mostly affected by the actions of other operations. For example, a CALL is equivalent to an 'LD (SP),PC' and two 'DEC SP's'. The PUSH, POP, RET and RST instructions also change it automatically. There are nevertheless a few instructions that manipulate it directly; ADD HL, DEC, INC, LD and exchanges with HL, IX and IY. The values held in the stack pointer, and its current location, is of vital importance to the programmer.

The 8-bit I and R registers; Interrupt vector and Refresh, are used to read the interrupt state, and are of limited importance. It is possible to LD I,A; LD A,I; LD R,A; and LD A,R only.

The PC register or program counter is a 16-bit single register. It cannot, unfortunately, be manipulated directly, its contents depending on the instruction being executed. It is incremented automatically as a pointer to the next instruction, and can be loaded with different values using CALL, RET, RST, RETI, RETN, JR and JP. A JP 5000 is equivalent to 'LD PC,5000'.

2.2 FLAGS AND CONDITIONS

The Z80 has six 'flags' in the register denoted 'F' or flags register. A flag is a bit that is set or reset according to the result of some operation. Without flags the computer would be unable to make decisions and be practically useless. The purpose of this section is to explain how flags are affected by various Z80 operations.

Since flags are made up of bits, many of their functions depend solely on bit and logical operations, which are also bit orientated, and are therefore difficult to explain separately. If you are unfamiliar with the more complex aspects of flag and bit operations, it is important that you read over this section and the next several times.

The F register has the following format:

BIT	SHORTHAND	NAME
0	C	Carry
1	N	Add/Subtract
2	P/O (P/V)	Parity/Overflow
3	—	Unused
4	H	Half-carry
5	—	Unused
6	Z	Zero
7	S	Sign

Some flags are used constantly, while others are used hardly ever or never. A compare operations only function is to set a flag condition. The programmer should be familiar with:

```
CP      50H

JR      Z,TRUE
```

If the value in the accumulator equals 50H, a jump is made to TRUE.

A compare operation actually subtracts 50H from the accumulator, but discards the new result. If the discarded result of the compare is zero, the previous value must have been 50H, and the 'Z' flag is therefore set. (Confusion can reign here. Remember: the Zero flag IS SET when bit 6 of F is a '1'. Therefore, Z is *set* if equal, *reset* if unequal.) The value in the accumulator is not changed by the compare.

The CP instruction can only be used for 8-bit values. What if we wanted to compare HL to DE? The following routine does this:

```
CMP     LD      BC,0        ;CLEAR
        LD      A,H         ;COMPARE HIGH
        CP      D
        JR      NZ,CP2      ;NOT EQUAL?
        LD      A,L         ;COMPARE LOW
        CP      E
        JR      NZ,CP2      ;NOT EQUAL?
        SET     6,C         ;SET BITS
BACK    SET     0,C
BACK1   PUSH    BC
        POP     AF          ;PUT IN AF
        RET
```

```
CP2     JR      NC,BACK      ;SET IT
        JR      BACK1
```

It compares H (the most significant register) to D and jumps to CP2 if unequal. A second check is made with L and E, jumping it sets bit 0 of C and bit 6 of C, exchanges C with F and RETurns. CP2 checks to see if the number is greater or smaller and sets or resets the bits accordingly. The possible conditions returned are Z, NZ, C and NC.

The routine is large and cumbersome, but one of the few options available if CP were the only instruction that enabled the setting of conditions. Fortunately, flags are set automatically by some other instructions. To check, conventionally, for zero:

```
CP      Ø
JR      Z,YES
```

However this could also be done by:

```
OR      A               ;OR "AND A"
JR      Z,YES
```

This is because the OR A instructions (which incidentally does not change the contents of the accumulator) sets the Zero flag if zero, just like CP, but takes a byte less to execute.

Instructions may set or reset a flag, have no effect, or set a flag on a condition; such as being equal to something. Here is a better 16-bit compare routine:

```
PUSH    HL
OR      A               ;CLEAR CARRY
SBC     HL,DE
POP     HL
```

The SBC automatically sets the Z, NZ, C and NC flags according to the result. The PUSH and POP instructions do not affect F. As demonstrated, a complete understanding of the Z80 makes programming a lot easier.

FLAG CONDITION	MEANING
Z	the value compared is equal
NZ	the value is unequal
C	the accumulator is less than the value
NC	the accumulator is greater than the value
PE	an overflow was produced
PO	no overflow was produced
M	the accumulator's *signed* value is less than
P	The accumulator's *signed* value is greater than

FIGURE 2.0 – FLAG CONDITIONS & MEANINGS

Refer to figure 2.0 to examine all possible flag conditions. The conditions Z, NZ, C, and NC should be straightforward. For example:

```
CP      50H
JR      C,SMALL
```

Will branch to SMALL if the value in the accumulator is less than 50H.

```
CP      50H
JR      NZ,NOEQU
```

Will branch to NOEQU only if the accumulator is *not* 50H. The conditions M and P depend on the *signed* value of the number. A signed number reserves bit 7 as a positive or negative bit. If bit 7 is set the programmer treats it as a negative number, otherwise positive. In this case bytes have a range between -128 and 127, instead of zero to 255. It is more common to treat numbers in the 'normal' positive fashion, so M and P are not 'popular' flags. As well, the Carry flag can sometimes serve the same function as M or P (to detect an *overflow* – which changes sign anyway).

Flag conditions C and NC can be used as smaller than, and greater than or equal, indicators; but how? A *carry* occurs when an addition or subtraction cannot be

contained in the normal limits of a single byte (0-255):

```
LD      A,ØFFH          ;SIGNED -1
ADD     A,1
```

When the addition takes place, the accumulator 'clocks around' and returns to zero. Adding one to 255 yields zero and a carry. A 'carry' is an extra bit – a kind of bit 8 – that goes into the flags register instead of the byte. Examine this in binary:

```
  11111111B
        1B  +
  ----------
 /10000000B
```

This last bit cannot be held in the normal limits of a byte. Observe a more complex example:

```
  11011101B          (0DDH)
  11101100B  +       (0ECH)
  ----------
 /11001001B          (0C9H) + carry
```

The 'carry' is therefore moved into the flags register. Examine the following compare:

```
LD      A,ØDDH
CP      ØECH
JR      C,TRUE
```

A jump is made to TRUE since there was a carry. The carry occurred because the value in the accumulator was smaller than the value being compared. The subtraction in the compare caused this carry. Any addition that causes a carry really acts as a subtraction, hence we can 'pretend' that numbers are signed because they behave in this manner anyway. Adding 255 to one, yields zero. The 255 seems to behave, in this case, as if it were a negative number. It is extremely important

that you understand all these relationships. Any missing information will limit your understanding of programs and routines in this and other books.

The PO and PE conditions serve a dual purpose. Their first is to detect signed *overflow*. This should not be confused with carry, both are distinctly different (although – sometimes – both conditions are interchangeable). We can use carry to detect if a number has exceeded its unsigned range, so too can we use overflow to determine signed ranges. Overflow will occur if the result of an operation is below -128 or above 127. Such that:

```
LD      A,127          ;MAXIMUM
INC     A              ;OVERFLOW
JP      PE,TRUE
```

Will create a signed overflow. For 16-bit numbers the range is -32768 to 32767. Its second function (the reason why it's given two initials, P/V – the initial 'V' is used instead of 'Overflow' to avoid mistaking it with zero), is a test for *parity*. It is used to detect whether the number of set bits in a byte are even or odd. Since there is always the chance that data will be lost travelling along a port, parity serves as a validity check. If a peripheral sent off four bits of data, but the parity was odd, the program would be able to detect a faulty transmission. Obviously it is not perfect – the scrambled data might be even as well, but it does help to make the check anyway.

Avoid the impression that there is any relationship between the number of bits in a byte and a numbers even or oddness. There is not. For example, the odd number 15 (00001111B) has an even number of bits. The odd number 31 (00011111B) has an odd number of bits. (There are, however, other ways to test if a number is even or odd.) Parity applies to I/O instructions, some shifts and rotates, and the logical instructions. Be careful that the instruction you use genuinely affects this flag. The following is a summary of the important functions of each flag:

Carry (C)
The carry bit serves as an indicator for addition and subtract operations, functioning as an '8th bit'. Subtracting 50 from a register that only holds 40, for example, will cause an overflow and the carry will be set. Therefore, carry can be used to determine if a value is greater or smaller than the compared value. Carry is used in SBC (SuBtract with Carry) and ADC (ADd with Carry). It is also used in shift and rotate instructions as an extra bit, as in the use of RRCA (Rotate Right with branch Carry of A). The Carry is affected by the result of all arithmetic operations and rotate instructions. All logical operations reset the Carry.

In Z80 mnemonics it is either a C (carry) or NC (no carry).

Add/Subtract (N) & Half-Carry (H)
Rarely used by the programmer. It is applied in Binary Coded Decimal Operations (a relatively rare application). There are no Add/Subtract conditional jumps.

Parity/Overflow (P/O) or (P/V)
This flag serves as a 'Parity' bit and an 'overflow', a type of signed carry. Parity is calculated by counting the number of bits set in a byte. If this is odd, the parity is reset. This register also has several other specialized functions. It is used to detect whether BC is zero during a block move instruction (for example LDIR) or block compare (for example CPIR).

In Z80 mnemonics it is either a PO (no overflow or parity odd) or PE (no overflow or parity even).

Sign (S)
The sign bit is affected by bit 7 of the byte tested. This is because, as briefly discussed, bit 7 can be used to represent the positive or negative value of a byte.

In Z80 mnemonics it is either an M (signed value less than) or P (signed value greater or equal).

Zero (Z)
This flag is used to determine if the value of a byte is equal or unequal. It is used by compare instructions to indicate match and by bit instructions to check if a bit is set or reset.

In Z80 mnemonics it is either a Z (equal) or NZ (not equal).

The appendix section of this book gives a flag summary. Try to learn it completely or the more important parts of it.

In summary, for conditional branches, the following symbols represent the flag:

Z	=	Zero (Z set)
NZ	=	Non Zero (Z reset)
C	=	Carry (C set)
NC	=	No carry (C reset)
PO	=	Parity odd or no overflow (P/V reset)
PE	=	Parity even or overflow (P/V set)
P	=	Positive (S reset)
M	=	Minus/negative (S set)

Now that we have a general understanding of the principal functions of each flag, some of their applications can be directly demonstrated.

Smaller than:

```
JR      C,TRUE
```

Smaller than or equal:

```
JR      C,TRUE
JR      Z,TRUE
```

Equal:

```
JR      Z,TRUE
```

Unequal:

```
JR      NZ,TRUE
```

Greater than or equal:

```
JR      NC,TRUE
```

Greater than:

```
JR      Z,$+4
JR      NC,TRUE
```

Miscellaneous conditions include:

```
BIT     7,A         ;TEST BIT 7
JP      NZ,TRUE     ;IF '1' JP
```

which could be written as:

```
OR      A           ;SET FLAG
JP      M,TRUE      ;IF '1' JP
```

Alternatively:

```
BIT     0,A         ;TEST BIT 0
JP      NZ,TRUE     ;IF '1' JP
```

could be written as:

```
RRCA                    ;BIT 0 TO C
JP      C,TRUE          ;IF '1' JP
```

This however changes the value of A, whereas BIT does not. Rotate instructions will be dealt with in the next section.

An odd number always has bit 1 set. To test:

```
BIT     0,A
JR      Z,ODD
```

or to make a number of unknown value even by decrementing:

```
RES     0,A
```

Test for 255:

```
INC     A
JR      Z,TRUE
```

although the value in the accumulator is changed. To test a register:

```
INC     B
DEC     B
```

Register B is used but any general purpose register can be tested in this way.

In block compares the zero flag is set if the correct byte is found, otherwise reset if the program decremented BC to zero. Moreover, JP PE indicates that BC has not been decremented to zero. (Since the Z flag is already set according to the compare condition, the P/V flag is reserved for the BC condition.) The function of the carry flag should not be misinterpreted. For example:

```
LD      HL,5000

LD      BC,6000

AND     A

SBC     HL,BC           ;CARRY
```

has done two things. It has set the carry and sign flags such that:

```
        JP      C,TRUE
```

or

```
        JP      M,TRUE
```

will transfer control to TRUE. Carry can be used to monitor a division operation. If HL is being divided by DE, an overflow will occur when HL has passed zero.

```
SUBTR   OR      A
        SBC     HL,DE
        JR      C,DONE          ;CARRY!
        INC     B               ;QUOTIENT
        JR      SUBTR
DONE    ADD     HL,DE           ;GET REMAINDER
        RET
```

Although JP M,DONE could have been used, this would have taken an extra byte since there is no 'JR M,nn' (Jump relative) version of the instruction.

2.3 WORKING WITH BITS

Working with bits is not difficult, but it does require some extra effort since many of these 'methods' are not visible to even a machine language programmer. It is also harder for us humans to relate to, as we are used to performing operations on whole numbers and not subdivisions.

Bit calculations can be divided into 'logical operations', 'rotates and shifts' and 'bit manipulations'. Since their application is extensive in this book, and a proper understanding of them is often limited – *even for programmers who have published software commercially* – the topic will again be covered here.

In most applications bit operations are rarely used, simply because there is no call for them. Or in some cases, misused, especially when a simple SUB or ADD instruction would do. Bits have very specialized functions and are limited in scope, but can, and often are, used in situations which prove ideal. Working with bits may be the only way to generate an attractive graphic effect (or at least one of the highest standard), and therefore has hardware importance: depending on the computer being used. Perhaps best of all they can be applied as powerful compare operations that allow the programmer to test, sort and search whole sets of data at once.

The Z80 allows us to turn bits on or off, or, test for the two conditions. Bits can be shifted or 'slide' left or right, or rotated around in a loop. Lastly, the programmer can add or remove them depending on its relationship with other bits in other bytes. Unfortunately, doing all this requires you to work in binary (a number base system programmers have been trying to escape from since the computer was first invented).

The only way to ease the rigmarole is to develop an efficient method of hexadecimal to binary (and vice versa) conversion, which will be dealt with in the final section of this chapter. Keep in mind that an AND 31 or XOR 7FH means nothing in hexadecimal, but in binary it is possible to 'follow along' with the computer program. So in reality, logical operations don't just appear confusing, but are confusing – until we see them in binary.

BIT MANIPULATIONS. As you know, a byte is made up of 8 bits (BInary digiTS). Hexadecimal was selected as the number base system that programmers would use since it is relatively easy to convert between these two bases. The hexadecimal number 0FFH can be represented in binary as 11111111B (remember the 'B' appended on the end of the number is the assembler code for binary). All bits are set to 'on' or '1' and represent the maximum binary number that can be contained in 8 bits. You should already be familiar with the binary number system, just as you are familiar with hexadecimal, so there is no need to go into too much

detail regarding the representation of numbers, but it is important to examine the layout of bits in a byte.

The leftmost bit position of a byte is called 'bit 7', the rightmost, 'bit 0'. So in the binary number 0100000B, only bit 6 is 'set'. When a bit is 'on' or '1' it is termed *set* and when zero, termed *reset*. In Z80 bits can be turned on and off using the SET and RES instructions:

```
LD      A,0FEH          ;11111110B
SET     0,A
```

will set bit 0 and the value of A will change to 0FFH (11111111B). Conversely:

```
LD      A,0FFH          ;11111111B
RES     0,A             ;RESET A
```

will reset bit 0 and the value of A will change to 254 (11111110B). Bits can be tested to see if they are set or reset. Called a *test* operation, the correct Z80 mnemonic is 'BIT':

```
LD      A,0FFH          ;11111111B
BIT     5,A
JR      NZ,YES          ;SET?
```

The condition is true since all bits are set. You will probably already be familiar with these operations, but are not sure how to use them. An immediate application is to check whether a signed number is positive or negative by doing a BIT 7,A. Bits are ideal for storing simple yes/no type data. Instead of storing away eight bytes, a positive or negative response can be kept in each bit of a single byte. Let's conjecture that a certain computer function can only be carried out after four other conditions are met. (It may be four prerequisites for joining a club, or getting credit on an account, or just four subroutines in a program that must be executed before the next one can function to satisfy a runchart.) Using bytes, four compares can be made:

```
LD      A,(CON1)        ;CONDITION 1
CP      1
RET     NZ
```

```
        LD      A,(CON2)    ;2ND CHANCE
        CP      1
        RET     NZ
        LD      A,(CON3)    ;3RD
        CP      1
        RET     NZ
        LD      A,(CON4)    ;4TH
        CP      1
        RET     NZ
SUBR4   . . .
```

Which was a lot of code to say the least. What if they had been stored away as bits? There is a maximum of eight yes/no conditions available in each byte. If the condition is *only* valid when the first four conditions are met. Then:

```
        LD      A,(CON1)
        CP      0FH         ;00001111B
        RET     NZ
```

checks for all four conditions simultaneously since 0FH is the only number with the left most nibble set. (A nibble is half a byte, made up of 4 bits. The right nibble is made up of the rightmost 4 bits and is sometimes called the MSB nibble, while the left nibble is made up of the leftmost 4 bits and is sometimes called the LSB nibble.)

This is all very well when four conditions apply to the routine, yet making a check for these 4 bits – when other bits are set as well – is not as straightforward. Say for example all eight conditions are true (11111111B), but you are only checking for the leftmost nibble (00001111B). It makes no difference to you whether the rightmost nibble is set or not. The other bits have changed the hexadecimal equivalent of the byte to 0FFH instead of 0FH, making the CP 0FH invalid. To solve this problem we could use consecutive RES instructions to zero out the right nibble before the compare, but there is an easier way.

LOGICAL OPERATIONS. These are not defined in the English terms 'AND', 'OR', and 'NOT'. While the Z80 has the instructions AND and OR (as well as a special OR called 'XOR' or exclusive OR, otherwise known as EOR), it lacks the NOT instruction. This, however, can easily be implemented using other instructions.

Logical functions are normally used to blend two sets of bits together; either adding or excluding certain bits according to the operation. They only affect bits relative to their position in bytes. Bit 5 of the first operand, will only affect bit 5 of the second operand and so on. Logical operations never set bits unless the bits were already set in one of the operands. The carry flag is always cleared by an AND A or OR A instruction(reset) and is affected by the outcome of various other operations.

The AND instruction is mostly used to remove certain bits from bytes (called *masking).* The rules for the AND instruction are: 0 AND 0 = 0, 0 AND 1 = 0, 1 AND 0 = 0 and 1 AND 1 = 1. Therefore a bit will only be set in the result if both bits from the operands were already set:

accumulator	=	00001000B
operand	=	00100000B
result	=	00000000B

while:

accumulator	=	01000000B
operand	=	01000000B
result	=	01000000B

thus:

accumulator	=	01001010B
operand	=	10111000B
result	=	00001000B

If we wanted to mask out the rightmost nibble (which is the problem we had before) we could do this by:

```
LD      B,0FH           ;00001111B
LD      A,(CON1)
AND     B
```

The rightmost nibble will be reset, regardless of its previous condition. *Only reset bits* will affect the result of an AND operation. The golden rule for this operation is: Wherever a zero occurs in the second operand, the corresponding bit will also be zero. Wherever a '1' occurs in the second operand, the corresponding bit will remain unchanged.

To make sure you have the knack of it, ask yourself this question: If I load the accumulator with 10000000B, what will happen to the number if I AND 0FFH?

Using the 'data storage' principle again, we could determine what two numbers have in common (the number could represent two people, two cheque accounts, etc.) by ANDing it. Say we wanted to set up a computer dating business (there are a million examples). Listed in the first byte of information about each person are eight yes/no questions about, say, drinking. By ANDing person A with person B, and then counting the number of set bits left, we will be able to evaluate their compatibility. Trying to do this using normal compares would be a horror.

Rather than being selective by setting a bit only if both original bits were already set, OR will set a bit if either of the two are set. The rules for OR are: 0 OR 0 = 0, 0 OR 1 = 1, 1 OR 0 = 1, and 1 OR 1 = 1. The operation is called an 'inclusive OR' and works as follows:

```
accumulator     =     00010000B

operand         =     00010000B


result          =     00010000B
```

while:. . .

accumulator	=	01000000B
operand	=	00000100B
result	=	01000100B

thus:

accumulator	=	10011001B
operand	=	00100100B
result	=	10111101B

which combines all the bits from the accumulator and the operand, placing it in the result.

We saw how we could use the AND instruction to mask out bits by resetting them. Using OR we can set bits. Using the 'data storage' principle again, there might come a time when four conditions (or any number of conditions) are met at once. Rather than individually setting these bits using the SET instruction we could:

```
LD      B,0FH           ;00001111B
LD      A,(CON1)
OR      B
```

which would effectively SET the left nibble. The golden rule for the OR instruction is: Wherever a '1' appears in the operands a '1' will appear in the result.

One of the more common uses of an OR is to check a 16-bit register pair for zero. This could be done by:

```
LD      A,B
OR      C
```

To make sure you understand the principles involved in using the OR instruction, ask yourself what number would be held in A if BC equalled zero after executing the latter two instructions.

The XOR instruction is the most fussy. An 'exclusive OR' is similar to an OR instruction except when both operands have the same bits set, which results in a cancel. The rules for XOR are: 0 XOR 0 = 0, 0 XOR 1 = 1, 1 XOR 0 = 1 and 1 XOR 1 = 0. The XOR instruction will still merge bytes together but will exclude pairs of bits. Such that:

accumulator	=	00001000B
operand	=	00000010B
result	=	00001010B

while:

accumulator	=	10000000B
operand	=	10000000B
result	=	00000000B

thus:

accumulator	=	01011001B
operand	=	01100101B
result	=	00111100B

In 'data storage' XOR could be used to eliminate similarities. It is often used by programmers as a 'toggle switch'. If you wanted to execute a routine every second time it was CALLed (useful in flashing displays, etc.) you could:

```
LD      A,(COUNT)    ;HOLDS SWITCH
XOR     1            ;PRESS SWITCH
LD      (COUNT),A
CP      1
RET     NZ
```

The location COUNT is initially loaded with one. When XORed by one, the two ones cancel out and a RETurn is executed. Executed the second time, COUNT has a value of one, hence only one of the operands now holds a one. The routine will be executed. The process continues with COUNT continually switching between one and zero. Exclusive OR is also an efficient way of clearing the accumulator; XOR A. The golden rule for XOR is: wherever a one appears in either operand a one appears in the result, except when corresponding bits are set.

The resetting of bits that are set is called *complementing;* which means inverting bits (setting what is reset and vice versa). The only purpose NOT has is to complement bits. The CPL instruction (short for ComPLement) does this, but is not a logical operation. So it doesn't affect the conditions of flags like other logical operations would. Another way is to utilise the XOR instruction. An XOR 11111111B or for short, XOR 255, will do the same thing as NOT. The rules for NOT are: NOT 0 = 1 and NOT 1 = 0. Such that:

accumulator = 01011001B

result = 10100110B

The NOT instruction is useful for generating opposites and reversing data. The golden rule for NOT is: A set bit is reset, a reset bit set.

ROTATES AND SHIFTS. These are primarily used to multiply and divide registers by two, align data or check bits, which can all be done using other instructions. However it is usually easier to take advantage of these operations. Rotate instructions (like shift instructions) move the bits within a byte from one bit position to another right or left. Each instruction can only move bits over one position at a time. For example, 01000000B rotated right, would change to 00100000B.

Rotate instructions differ from shift instructions because they 'wrap-around' bits. If we rotated a byte left, bit 7 would replace bit 0, and alternatively rotating right, bit 0 would replace bit 7. If we shifted 10000001B one position to the left, Bit 0 would 'fall off the end' so to speak, and the new value would be 01000000B. In a rotate it would swing around, and the new value would be 11000000B.

The major rotate instruction subset includes:

RCLA	---->	Rotate A left circular
RLA	---->	Rotate A left
RRCA	---->	Rotate A right circular
RRA	---->	Rotate A right

The registers A, B, C, D, E, H, L, and (HL), (IX + d) or (IY + d) can also be rotated, but in mnemonic form there must be a space between the instruction and operand (for example RR E, RL C, etc.). The difference between 'Rotate A left circular' and 'Rotate A left' is probably not clear so I will explain further.

Rotate and shifts are tied up intricately with the carry flag. When an RRCA is performed, the content of bit 0 is moved to bit 7 as well as the carry flag. A programmer could therefore check if bit 0 of a byte was set by:

```
RRCA

JR      C,SETON
```

Provided that bit 0 of the byte was set, carry will result. I previously talked about ANDing person A with person B in a computer dating program and then counting the number of views that corresponded. This addition could have been done by executing eight BIT instructions. But by rotating a byte around (or shifting) eight

times in a loop, the carry flag need only be checked. This short routine illustrates the principle:

```
        LD      A,(COMMON)
        LD      B,8             ;COUNTER
        LD      D,0
LOOP    RRCA
        JR      NC,SKIP         ;NOT SET?
        INC     D
SKIP    DJNZ    LOOP
        RET
```

The RRA instruction on the other hand uses the carry flag differently. While Bit 0 is still shifted to the carry, the flag's previous value is moved to bit 7. This is called a 9-bit rotation since one byte plus the carry is moved.

If you need to set all flags (not just carry) you must perform an ADC A,A instead, which still rotates the bits like an RLA instruction. Diagram 2.1 illustrates these rotate operations.

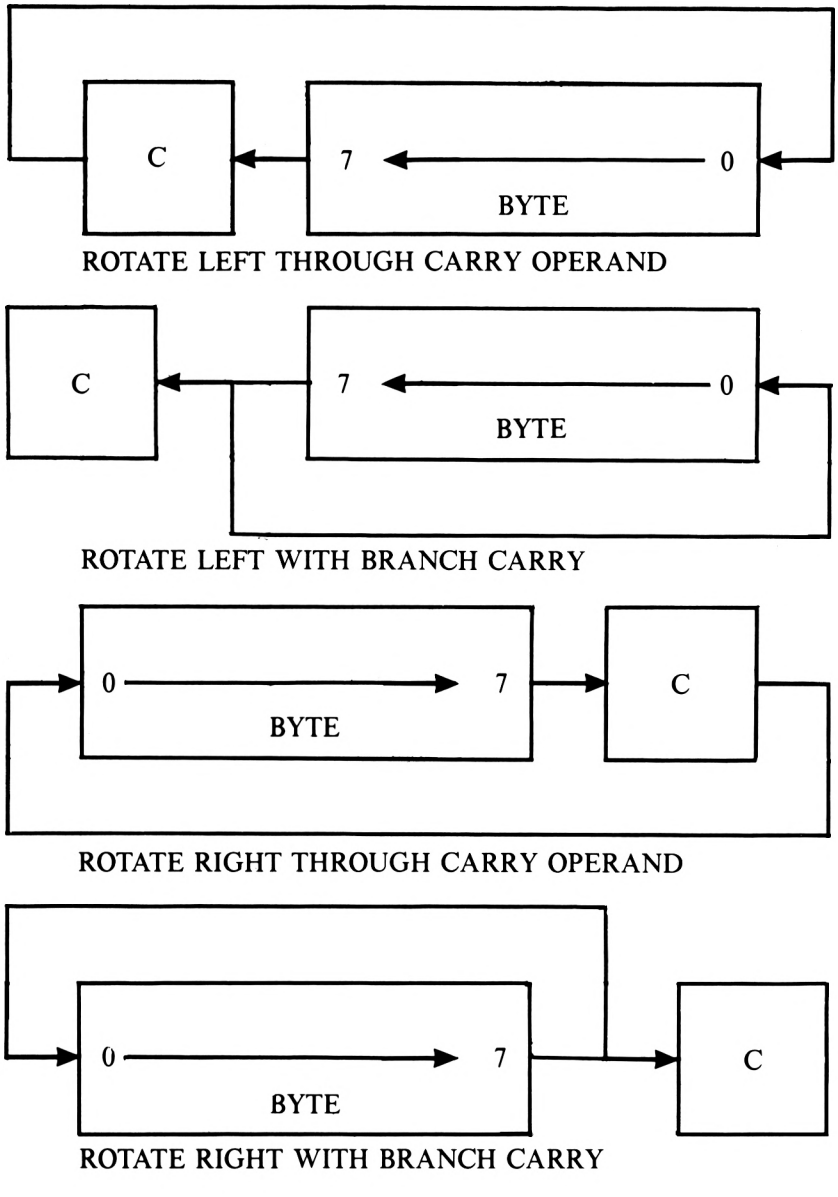


DIAGRAM 2.1 – ROTATE OPERATION GROUP.

The major shift instruction subset includes:

SLA r	----->	Shift left arithmetic
SRA r	----->	Shift right arithmetic
SRL r	----->	Shift right logical

Where r is a register A, B, C, D, E, H, L, or (HL), (IX + d) or (IY + d).

In an SLA operation the bits of the byte are shifted one position left. Bit 7 is copied into the carry flag and bit 0 is reset.

In an SRA operation the bits of the byte are shifted one position right. The content of bit 0 is moved to the carry flag, and bit 7 remains unchanged.

In an SRL operation the bits of the byte are shifted one position right. The content of bit 0 is moved to the carry flag, and bit 7 is reset. Diagram 2.2 illustrates these shift operations.

Shifts are useful when multiplying and dividing by two. This process is handy for doubling or chopping numbers in half. An SRL A instruction will divide the A register by two. While ADD A,A will multiply the accumulator by two, an SLA instruction can multiply any register. Shifts and rotates present no problem when dealing with 8-bits, but become tricky with register pairs. The following commands will achieve the desired results:

Shift HL left logically:

```
ADD     HL,HL
```

Shift HL right logically:

```
SRL     H
RR      L
```

Shift HL right arithmetically:

```
SRA     H
RR      L
```

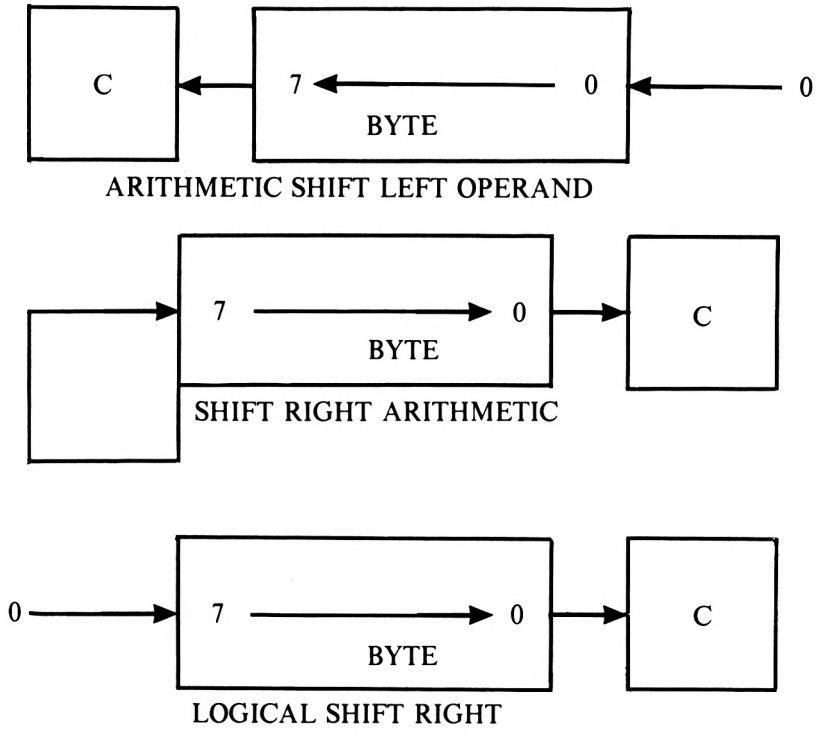


DIAGRAM 2.2 – SHIFT OPERATION GROUP.

Rotate HL right:

```
RRC   L
RL    L
RR    H
RR    L
```

Rotate HL left:

```
RLC   H
RR    H
RL    L
RL    H
```

Although HL is used in each example, BC and DE can also be shifted or rotated in a similar manner. Such manipulations effectively divide and multiply register pairs by two.

One or two other instructions work with bits, but their purposes are so specialized that they are not covered here (refer to miscellaneous instructions). Learn how to apply shifts and rotates by examining their functions in other routines. There are a number of good examples in chapter three and other listings throughout this book.

Some programmers hardly ever use bits, but their coding is also larger and more clumsy.

2.4 SPEED VERSUS MEMORY

Unless you intend number crunching to several hundred decimal places, bit shifting 5K of RAM, or sorting two thousand records, most programs in Z80 are executed in a reasonable period of time. This varies with the clock speed of your system (anything from 1-5 MHz for Z80As and even higher speeds for higher quality Z80s). Even a computer running at under two MHz is capable of some very high speed operations at very close to a quarter of a million instructions a second – although you should take speed limitations into consideration and not get unduly carried away by these figures. Counting the number of instructions needed to do even a simple task, quickly pales the sort of heady speeds attributed to computer processing.

Frankly, low level coding is difficult enough without having to worry about an instruction's 'M cycle' or 'T state'. A conscious, clean, well designed routine is the fastest, and the most memory conservative way of programming. Any other method of 'speed' programming is just not realistic. In the long run the algorithm used is more important in determining the speed of execution, rather than individual instructions.

If you still feel strongly about writing fast code, which you may feel justified in subroutines that are repeated thousands of times (such as sorting) then the best general advice is to avoid all 'new' Z80 instructions; that is, instructions that were not available on the earlier 8080 microprocessor. Those that begin with 0EDH, 0CBH, 0DDH or 0FDH. They include many shifts, the index registers, and a variety of other special instructions. The block move and block compare group, although 'new' are still fast and efficient enough to warrant use.

Small code is inherantly fast, since there are usually fewer instructions to execute. A smaller program means less typing and debugging and more memory. During

coding check to see that you're taking advantage of the many aspects of the Z80 that make your programs smaller and faster. The following is a small list of general observations:

- Have registers been appropriately utilized (as discussed at the start of this chapter) to their best advantage? Register swapping could indicate a bad initial choice.

- Have you loaded your registers before entering loops? If you need only make a compare once, then put that compare outside the loop. Rather than continually loading a register with a memory address. It is faster to have a register reserved as a constant.

- Kill two birds with one stone: why not load two 8-bit registers using a single 16-bit load operation? Working in hexadecimal this is easy enough to calculate.

- Take full advantage of advanced features of the Z80, like block move and block compare operations where practical.

- Check to see that you're getting the best out of your program's organisation. If you're making only one compare, you should need only one label to jump to. If you need more, reorganise your program (this is probably a case of changing Zs to NZs, etc.).

- Place data in easy to access index tables. Even if your data is instruction code. Rather than jumping to five or six small subroutines, execute a loop that will place the appropriate code in a certain location and then execute it. This is practical when routines differ only slightly with one another.

- Take advantage of operations that work on data directly in memory. Using instructions like INC (HL) there is no need to store a final result away.

- Preferably PUSH a value on the stack temporarily, instead of saving it in memory. Instead of storing HL on the stack and then getting another value off, use the EX (SP),HL instruction to make a quick and efficient exchange.

- When checking for flag status, place the jumps that are likely to happen more often before the less common possibilities.

- Since you have to clear the carry (by an OR A or AND A) before subtracting a 16-bit value, add a negative (signed) number instead.

- Avoid complicated shifts or logical operations when a simple add or subtract will do (although this can be easier said than done).

- Try not to move blocks of data around your program, as this can be 'messy' organisation; readjust pointers instead.

- Take full advantage of available ROM routines when suitable. Note however that these routines may do a lot of 'house keeping'; things important for ROM, but not for your program. Thus they might be slower than versions you could write.

- Instead of PUSHing and POPing general purpose registers, use an EXX instruction to swap register sets when there is no need to pass parameters.

- Determine if you are taking full advantage of all flag conditions. A flag could automatically be set according to an instruction you *could have* used, or there may be an unnecessary compare statement (flag already set by a previous instruction).

- Choose rotates and shifts carefully. Try and make the accumulator the register to shift/rotate. Accumulator rotates are faster and use less memory. As well, you can AND any unwanted bits without further exchange.

The programmer must always be conscious of the amount of memory consumed by the editor/assembler program. Remember that a line number takes between three to five bytes of source code to keep track of, so select things like ADD A,2 instead of two INC A instructions. MACRO, or preferably CALL instructions should be utilized as much as possible. Unfortunately, limited memory is the best encouragement for writing compact code.

Predefine commonly used constants such as the start of video RAM etc. with EQUates like VDU, J, VT etc. Avoid remark statements in large pieces of code, excluding even brief notes between subroutines. Don't follow the examples in this book. The comments are there to help you, but they should never be intermixed with actual assembly language. Most of your program should be extensively documented in your notebooks, not your source code!

2.5 MISCELLANEOUS Z80 INSTRUCTIONS

To tie up any loose ends and possible gaps in your knowledge, all instructions that tend to be fairly specialized and 'miscellaneous', are documented here:

CCF
Complement Carry Flag. The carry flag is inverted by this operation. If set, it will be reset and vice versa.

CPL
ComPLement accumulator. The contents of the accumulator are inverted. All set bits are reset and all reset bits are set. It is often used to change a positive number into a negative number, or vice versa.

DAA
Decimal Adjust Accumulator. The instruction adds six to the right and/or left nibble of the accumulator where appropriate for Binary Coded Decimal (BCD) operations. BCD calculations are rarely used, and are not discussed in this book.

DI
Disable Interrupts. Interrupts enable the computer to do several things at once, by 'interrupting' the main program, and performing another task before RETurning. This will slow any program down, so if interrupts are not required, they would be deactivated. It is therefore desirable to have 'DI' as the first instruction in a program. (Note that some computers do not have the necessary interrupt facilities to use some or any of these features – at least not without modification.)

EI
Enable Interrupts. Turns interrupts on.

EXX
EXchange alternate registers. The Z80 has a series of registers not directly addressable, denotated AF', BC', DE', and HL', called the 'alternate' set. When this command is executed, all general purpose registers are exchanged with this other set. Note that the value of AF is not changed. To swap back another EXX must be performed. This instruction is useful in self-contained subroutines, where no values need be passed (except anything in AF) to it. Rather than PUSHing all three registers, and then POPing, an EXX will keep working registers separate.

HALT
HALT CPU. The CPU repeatedly executes NOP instructions until an interrupt or reset is received. This has varying effects on computers, depending on internal layout. It could, for example reset the system. Which is equivalent to performing JP 0.

IM n
Interrupt Mode. Sets interrupt mode n (0-2) condition. Used only when working with interrupts.

IN A,(n)
INput port n placing result in accumulator. This is one method of receiving information from an I/O device (the other is memory-mapping). The instruction will check the port specified by n (from 0-255) and place a number in the accumulator in accordance with data sent by a peripheral.

LD A,I
The contents of the interrupt vector register are loaded into the accumulator. Used in checking the status of the interrupts.

LD A,R
The contents of the Memory Refresh register are loaded into the accumulator. This register is occasionally used in random number generating, but is otherwise of little use to the programmer.

NEG
NEGate accumulator. The contents of the accumulator are subtracted from zero. For example, NEGating one would yield 255. It is identical to the instruction SUB 255 and is the same number of bytes in length.

OUT (n),A
OUTput contents of accumulator to port n. This is one way of sending information to an I/O device. The instruction will send the contents of the accumulator to the port specified by n (from 0-255). If interfaced to this port, the peripheral device will receive the information.

RST n
ReSTart at n. The program counter is PUSHed onto the stack (just like a CALL instruction). It is then loaded with the address specified by n. The variable n can only equal 8H, 10H, 18H, 29H, 28H, 30H and 38H. Since this area is normally reserved as ROM, restarts are not normally accessable to the programmer. In some cases, ROM programmers have allowed RSTs to be vectored into RAM, making it possible to access them (by changing the jump back to ROM to another address). In almost all respects, RSTs are one byte CALL instructions.

A word of caution should be added before using RSTs. Since they are almost always used by ROM programmers, revectoring may disable (temporarily of course) some of the functions of these ROM programs. The programmer needn't worry about possible 'side effects' unless using ROM routines.

Hopefully you now have an understanding of all Z80 instructions. (With the possible exception of block I/O and interrupt handling. For more information on these topics refer to books concerning Z80 interfacing.) Finally, I recommend you purchase a well documented Z80 instruction reference book. It will give details concerning function, format, data flow, timing, addressing mode, byte codes, the effects on flags, and examples; besides the instruction description. Even the veteran Z80 programmer will find the occasional need to refer to one.

2.6 WORKING WITH A CALCULATOR

Binary, decimal and hexadecimal base conversions are essentially easy to do using a computer, but it is likely that you will not have a computer, or the computer will be occupied with some other task when the need arrives. Changing bases is particularly important, if not essential, when editing object code using a monitor program. The ideal solution is to purchase a programmer's calculator that has decimal, hexadecimal, binary and even octal conversion at the press of a button. It will also enable you to calculate the results of logical operations, and depending on the brand, multiple shifts and rotates. Calculators that don't work in binary should have a 4-bit binary number written under each alpha-numeric key. It will allow you to look at the binary and hexadecimal result at the same time. The only drawback is the price of the calculator, although it may be worth any price, depending on your need. If the relationship is casual, conversion on a normal calculator will have to do. A scientific calculator is handy (a scientific calculator that can handle bases is *even more handy)* though not essential.

Whether you are working with 45E8H, 643, or 101101B, these numbers all have one thing in common: each digit represents a power. If we studied the decimal number 123 we would see that the '3' digit is a power of '1' (1 * 3), the '4' digit a power of '10' (10 * 4), and the '6' digit a power of '100' (100 * 6). As the number adds more digits, so does the power increase by tens, because decimal is of course base 10. This rule applies to hexadecimal and binary numbers as well. As you may know, 0FFH is 255 decimal. Try adding (1 * 15) + (16 * 15). Remember: 0FH = 15. Multiplying by base 16 (the hexadecimal base) and you get 15 + 240 = 255. The algorithm is confirmed.

Binary, in principle, is exactly the same. The number 101101B is calculated using the formula: (1 * 1) + (2 * 0) + (4 * 1) + (8 * 1) + (16 * 0) + (32 * 1).

As we move across leftwise, each digit must be increasingly multiplied by 1, 2, 4, 8, 16, 32 . . . in other words, the last power multiplied by two.

Since a normal calculator cannot represent a binary or hexadecimal number it is more difficult to make calculations in these bases. An alternative is to get a decimal/hexadecimal/binary table counting from zero to 255, which is usually satisfactory for our 8-bit computers (such a table is in appendix C of this book).

A hexadecimal number can be split into two parts, a left and right digit. If we had to convert the hexadecimal number 20H to binary we would need to look up the binary equivalent of '2' which is 0010B and the binary equivalent of '0' which is 0000B. The two nibbles can then be joined together to produce 00100000B. See if you can work out what this number is in decimal by calculating the power. Doing this comes in handy, especially when trying to work out the bits for AND, OR and XOR instructions. If you easily came up with 32 you have just made programming a lot easier. This type of conversion is not difficult. In fact, the programmer should view hexadecimal as 'compact binary' and not an alien number system.

The last problem we are faced with is hexadecimal or binary to decimal conversion. The problem here is that decimal has nothing in common with binary or hexadecimal. To work out the binary equivalent of a number you must subtract the largest power of two. Since we are only interested in 16-bit numbers at the most, we have to keep in mind that:

1 * n has a power of 0
2 * n has a power of 1
4 * n has a power of 2
16 * n has a power of 4
32 * n has a power of 5
64 * n has a power of 6
128 * n has a power of 7

If you needed to convert 123 into binary you would search the table to find a number that is equal or less than it. The closest, without exceeding it, is 64, which has a power of six. Now subtract 64 from 123 to leave you 59. We now have the first part of our binary number: 1000000B + 59 = 123 (note that the '1' bit is placed in the seventh position of the binary number not the sixth, since you must also count the two times zero position).

The next closest power of two is 32. Which has a power of five. Place the '1' bit in the sixth position of the binary number to give you 100000B and subtract 32. You are left with 1000000B + 1000000B + 27 = 123. Now get the next number, 16, work out its power and add it to the equation to give you 1000000B + 100000B + 10000B + 11 = 123. The number eight has a power of three, such that 1000000B + 100000B + 10000B + 1000B + 3 = 123. The next (2) has a power of one, and the last (1) is a power of zero. Once you have worked out the

binary 'pieces' an addition gives the total:

```
  1000000B
   100000B
    10000B
     1000B   +
       10B
        1B
 ----------
  01111011B
```

Actually, as you can see, it is faster to do than explain.

It is nevertheless tedious to calculate. The quick way out is to look the hexadecimal number up in a tables section and convert that to binary. Which is fine if you have this book handy, or until you have to work out numbers over 255. In desperate situations you will have to go up to powers of nine.

Converting from decimal to hexadecimal relies on the same method as binary conversion. Here you will have to use a table with the values 1, 16, 256, and 4096. If we wanted decimal 15360 converted to hexadecimal, it would be necessary to calculate the number of times 4096 will go into the number. The maximum ends up being 3 * 4096 = 12288. This has a power of three, such that 3000H + 3072 = 15360 (we get 3072 by subtracting 12288 from 15360). The number of 256s is then calculated. This works out to be 12 since 12 * 256 = 3072. Or 0CH * 256 = 3072. The result is 3000H + 0C00H (a power of two) = 3C00H. Converting the number into binary is merely a matter of swapping appropriate binary nibbles for each digit.

These conversion principles work for any problem regardless of the base. Using a scientific calculator you can calculate the power of any number by pressing the 'powers' key. Using a normal calculator you will have to multiply by the base to calculate the next power.

Having completed structure, basic debugging, programming techniques, and base conversion, you are now suitably armed to go on to actual programming. If you don't feel confident about what you have covered, go through it again (but once more only), else, take a deep breath and turn the page.

CHAPTER 3 – GETTING THE MESSAGE ACROSS

Writing low level code requires you to solve problems not normally associated with high level programming. What is trivial to other programmers becomes a major problem in machine language. First of all, how are messages and numbers displayed efficiently on the VDU? Or for that matter, how is it possible to write a routine to read the keyboard for a string or a number? This chapter will solve these problems.

3.1 COMMUNICATING

Few peripherals, if any, are regarded as 'standardized' equipment, so I have limited this chapter to keyboard and VDU decoding, which is more than enough problem solving for a novice. Strictly speaking, even keyboard and VDU I/O is non-standard enough to present problems, but the routines demonstrated in this chapter are not particularly machine specific, so should be suitable for most applications.

Using the Z80, and fairly standard hardware, a computer is capable of displaying a maximum of 256 characters at a time (characters can be 'reprogrammed' using a programmable character generator if your computer has one). A 'character' is the symbolic representation of a number when displayed on the VDU. There are several 'standards' or orders in which characters are assigned to numbers. The one most commonly used, and the one used throughout this book, is ASCII.

ASCII stands for American Standard Code for Information Interchange, and most manufacturers – but unfortunately not all – have had the good sense to adopt it. And there are those computers that process ASCII, but have not strictly followed standard character organisation. Using a high level language there is no need to worry about this, as ROM makes the necessary calculations to adjust characters accordingly. Machine language may therefore be the start of your troubles.

Characters are not displayed randomly: the number assigned to character 'A' is one byte less than the number assigned to character 'B' and so on. The alphabet is in ascending order. The same applies to numbers. Refer to appendix D of this book, which lists the ASCII character set. You will notice that characters 0-1FH are 'control codes'. When a computer receives one of these it performs a special 'control' function. This might be tabbing, scrolling the VDU, backspacing, or some other function.

Some are dedicated to line printer control (and thankfully, most printers also respond to standard ASCII) and do not necessarily affect the VDU. More importantly, ASCII is an effective way of sending text (words and sentences) to other computers and peripherals. Hence, the 'information interchange' and the fundamental organisation of control codes. A fair portion too, are reserved for 'cursor' positioning. You are probably familiar with the thin line or blinking graphic block that marks your position when typing on the VDU.

Of course all this is controlled by software, since ASCII has no special significance, except when we programmers choose to give it some. Consequently most computer languages interpret some or all of these control codes, so that a code 1FH would instruct the computer (or more correctly; the computer language) to clear the VDU. The keyboard scanning routines in this book expect standard ASCII responses, particularly code 8 (backspace) and code 13 (return or 'carriage return', which not only instructs the cursor to move to a lower line or scroll the display, but would reset the carriage position on a responsive line printer).

Keep in mind that ASCII is an old code, and some of the names may seem a bit odd. For example, BEL actually *activated a bell,* and bi-directional printing was unheard of. Nevertheless, codes zero to 31 are still real characters, we just don't treat them as real characters when we use ASCII. What kind of characters are they? The less imaginative manufacturers have just repeated characters already existing in ASCII, or they may be used for inverted characters or other extra symbols not found in ASCII, which itself reserves a mere 63 symbolic characters.

Characters 20H ('SPACE' is a blank byte) to 7FH have no special control meanings and display alpha-numerics and special symbols on the VDU only. The LASER-200 character set is slightly different (as with a few others), having an alphabet starting at '0' instead of 41H. Instead of rewriting all the routines in this chapter (which is still practical, since the idea is more important than the code used) it would be possible to insert this small routine where appropriate:

```
        CP      41H
        CALL    NC,SOLVE
        .   .   .

SOLVE   SUB     41H
        RET
```

The routine checks to see if the number is equal or greater than 41H, and if true, subtracts 41H to align it with standard ASCIIs 'A' to 'Z'. In fact, this is what the LASER-200s ROM does to make its character set compatible.

It demonstrates that routines in this chapter are applicable to all Z80s using ASCII – with some fiddling.

To display a message on a VDU, three parameters are required. Where the message is located in memory, where it is going to be positioned on the VDU, and its length. Using a simple block move we could display a message using the following code:

```
;"VDU"  IS START OF VDU RAM
PRINT   LD      DE,VDU
        LD      HL,MESSG1
        LD      BC,11
        LDIR
        RET
MESSG1  DB      'GREY RODGER'
```

The block move is cumbersome and relatively inflexible. If you wanted to display messages at different locations on the screen you would have to reload all the registers and execute another block move. (I have presumed that your computer has a fairly typical VDU memory map. Refer to chapter five if your computer does not allow you to directly access VDU RAM.) The following routine can calculate spaces, allowing messages to be displayed in different locations while still using only one CALL:

```
;PRINT MESSAGE ROUTINE
;ENTRY -->    DE = MESSAGE
;             HL = VDU RAM TO START
```

```
PMG     LD      A,(DE)
        CP      80H             ;VALID ASCII?
        JR      C,PMG1
        SUB     7FH             ;GET OFFSET
        LD      B,0
        LD      C,A
        ADD     HL,BC           ;ADD IT
PMG2    INC     DE              ;NEXT CHARACTER
        JR      PMG
PMG1    OR      A               ;TEST ZERO
        RET     Z
        LD      (HL),A          ;DISPLAY IT
        INC     HL              ;NEXT VDU POS
        JR      PMG2

M1      DB      'LORD OF',100,'THE RINGS',0
```

Numbers above 7FH are treated as spaces to skip, byte zero is a terminator to be placed at the end of each piece of text. Using this routine, a single load and CALL can display a lengthy message at several locations on the screen at once.

To read ASCII from the keyboard (someone's name, for example) requires a routine that will 'string' alpha-numerics together into a buffer.

```
;INPUT ASCII STRING ROUTINE
;ENTRY -->    HL = BUFFER POINTER
;             DE = VDU POSITION

INPUT   LD      HL,BUFF     ;BUFFER
        LD      DE,PNT      ;DISPLAY
        LD      B,0         ;NO. CHARACTERS
LOOP    CALL    GKEY        ;GET CHARACTER
        CP      8           ;BACKSPACE?
        JR      Z,BACK
        CP      13          ;RETURN KEY?
        JR      Z,DONE
        PUSH    AF          ;SAVE AF
        LD      A,B         ;MESSAGE COUNT
        CP      MAX         ;MAXIMUM LIMIT
        POP     AF          ;RECOVER AF
        JR      Z,LOOP
        INC     B
        LD      (HL),A      ;SAVE CHARACTER
        LD      (DE),A      ;DISPLAY IT
        INC     DE          ;NEXT
        INC     HL          ;NEXT
```

```
        JR      LOOP
BACK    LD      A,B             ;MAXIMUM LIMIT
        OR      A               ;EMPTY?
        JR      Z,LOOP
        DEC     B               ;GO BACK
        DEC     HL              ;AGAIN
        LD      A,32            ;BLANK SPACE
        LD      (HL),A
        LD      (DE),A
        DEC     DE
        JR      LOOP
ENTER   LD      (HL),0          ;TERMINATOR
        RET

BUFF    DS      MAX
```

The label GKEY will either be the entry address of your own keyboard decoder or a ROM call (refer to the appendix section for this information). The BACK routine is a backspace function, moving pointers back to the last character. Remember that the codes '8' and '13' are ASCII FOR 'backspace' and 'return'. The label MAX is the buffer length, a variable between one and 255.

Essentially the routine reads a key, places it in the buffer and displays it on the VDU, then checks to see if the buffer limit has been reached. If it has, the routine ignores all further key strokes, except for backspace and return. Backspace deletes the last character by moving the buffer pointer back, and displaying a space over the previous character on the VDU. Pressing the return key (or a key that duplicates this function on your computer) loads the buffer with a zero, 'end of buffer' byte and the routine RETurns.

Inputting numerics is not as easy as inputting ASCII text. The programmer must check that the keys pressed are valid numeric input; digits '0-9' in decimal or digits '0-0FH' in hexadecimal. We will deal first with hexadecimal input. A one byte (8-bit) hexadecimal number is made up of two digits (the great thing about hex), the MSD (most significant nibble or right nibble) and LSD (least significant nibble or left nibble), while a two byte (16-bit) number is made up of four. By CALLing an 8-bit routine twice we can avoid having to write a special 16-bit routine.

Look over the following routine, keeping in mind that a hexadecimal number has the first digit in the left nibble and the second digit in the right nibble. When we input the first digit we must place it in bits 7-4, the second digit in bits 3-0:

```
;GET A HEXADECIMAL NUMBER FROM KEYBOARD
;EXIT  --->    A  = HEX NUMBER


GHEX    CALL  INPUT       ;1ST DIGIT
        LD    B,4
ROTATE  RRCA              ;SHIFT TO 7-4
        DJNZ  ROTATE
        LD    B,A
        CALL  INPUT       ;2ND DIGIT
        OR    B           ;COMBINE DIGITS
        RET
INPUT   CALL  GKEY
        CALL  SHOW        ;DISPLAY IT
        CP    30H         ;BYTE "0"
        JR    C,INPUT
        CP    47H         ;BYTE "G"
```

```
        JR      NC,INPUT
        CP      41H             ;BYTE "A"
        JR      NC,FIX          ;CORRECT "A-F"
        CP      3AH             ;NO GARBAGE
        JR      NC,INPUT
        SUB     30H             ;TO BINARY
        RET
FIX     SUB     37H             ;"A-F" HEX
        RET
```

The INPUT routine makes sure that only valid numbers are permitted. It then subtracts 30H or 37H where appropriate to convert the ASCII into hexadecimal. The first digit is shifted to the left using a right-rotate instruction, and the result (now in bits 7-4) is placed in B. INPUT is re-executed, and the B registers is logically merged into the accumulator using an OR instruction. The one byte result is left in the accumulator on exit. A 16-bit hexadecimal number could be input and placed in HL by:

```
HEX16   CALL    GHEX
        LD      H,A
        CALL    GHEX
        LD      L,A
        RET
```

The last thing that need be said about the routine is CALL SHOW. This routine displays the contents of the accumulator on the VDU. Either load a register like IX with the start of video RAM and:

```
SHOW    LD      (IX),A
        INC     IX
        RET
```

or use a ROM call if you know a suitable address.

While the subject of inputting hexadecimal numbers is fresh in your mind let us examine ways to redisplay them. This calls for a reverse of what we have already done, with bits 7-4 moved back to 3-0, and 30H or 37H added. A shift right-logical (shifting zeros into bit 7) instruction moves the nibble into position.

```
;DISPLAY A NUMBER IN HEXADECIMAL
;ENTRY -->    A  = HEX NUMBER

PHEX    PUSH  AF          ;SAVE DIGITS
        LD    B,4
SHIFT   SRL   A           ;RIGHT NIBBLE
        DJNZ  SHIFT
        CALL  PH2
        POP   AF
PH2     ADD   A,37H       ;BACK TO ASCII
        CP    3AH
        JR    NC,SKIP
        SUB   7           ;NOT "A-F"
SKIP    CALL  SHOW
```

On entry the accumulator holds the value to be displayed. The left nibble (bits 7-4) has to be moved to the right nibble (bits 3-0). Now it is a simple matter of adding 37H to get digits between '0A-0FH' and if less (digits '0-9'), subtract another seven. It can then be displayed using the CALL SHOW routine. To display numbers spanning 16-bits load the accumulator with H, CALL the routine, and repeat the process using L.

Binary to decimal conversion is in no way as straightforward. One conversion method adds or subtracts in multiples of 10, checking for a carry or decrementing a counter to terminate the loop. To display a decimal number between 0-99, the following routine subtracts in 10s and displays the remainder as the second digit:

```
;DISPLAY A 2 DIGIT DECIMAL NUMBER
;ENTRY --->    A  = NUMBER
;              IX = POSITION ON VDU
;
CONASC LD      BC,3030H     ;"0-9"
ASC    CP      10
       JR      NC,ASC1
       ADD     A,C          ;CONVERT "0-9"
       LD      (IX),B
       LD      (IX+1),A
       RET
ASC1   SUB     10           ;SUBTRACT
       INC     B            ;NEXT DIGIT
       JR      ASC
```

Initially BC is loaded with 30H (ASCII '0'), so when the accumulator is decremented by 10, B is incremented to the next ASCII digit. When finally less than 10, 30H is added to the second digit. The register IX must point to the area of VDU RAM where the number is to be displayed.

Displaying numbers larger than 99 requires a larger routine, since we have to subtract in 10000s, 1000s, 100s, 10s and finally 1s if we are going to display a number quickly. A 16-bit value has a maximum of five digits – with no number exceeding 65535. Rather than having five separate subroutines, it is possible to display the entire number using a loop:

```
;DISPLAY A 16-BIT DECIMAL NUMBER
;ENTRY -->    HL = NUMBER


DEC     LD     BC,500H      ;B=5,C=0
        LD     IX,DTAB
DEC1    LD     A,30H        ;ASCII
        LD     D,(IX+1)     ;GET NUMBER,
        LD     E,(IX)       ;TO SUBTRACT
DEC2    OR     A
        SBC    HL,DE        ;SUBTRACT
        JR     C,DEC3       ;A CARRY?
        INC    A            ;INC COUNTER
        JR     DEC2
DEC3    ADD    HL,DE        ;RESTORE VALUE
        INC    IX           ;NEXT NUMBER,
        INC    IX           ;TO SUBTRACT
        CP     30H          ;IS IT A "0"?
        JR     NZ,SKIP
        CP     C            ;AVOID LEADING,
```

```
        JR      NZ,PASS         ;ZEROS
SKIP    CALL    SHOW            ;DISPLAY IT
        LD      C,30H
PASS    DJNZ    DEC1
        RET


DTAB    DW      10000,1000,100,10,1
```

The register pair DE is loaded with the contents of DTAB, which is subtracted from HL (the holder of the 16-bit value to display). The accumulator is incremented, and the process repeats until a carry occurs. The previous value – before the carry – is restored, and the IX pointers are incremented to try to subtract a lesser value. A check is made to see if the accumulator equals 30H (ASCII '0') and if so, another check is made to determine if a previous number has already been displayed. This is important because it stops the routine from displaying leading zeros ('456' as opposed to '00456'). It then decrements the B register, and returns when all five digits have been displayed or passed.

Lastly, we have yet to input a 16-bit number from the keyboard. This is practically a reverse of the previous routine, instead of subtracting, we can multiply according to size and power of the digit. A simple routine to accept a decimal number would be:

```
GDEC    CALL    GKEY
        CP      13              ;RETURN
        RET     Z
        CP      30H             ;"0"
        JR      C,GDEC          ;BELOW "0"
        CP      3AH             ;ABOVE "9"
        JR      NC,GDEC
        LD      (HL),A
```

```
        INC     HL
        JR      GDEC
```

This makeshift input routine can be modified to suit your exact needs. The B register could be loaded with five (maximum digits allowed) and a DJNZ GDEC performed followed by a RET to stop HL form destroying the stack or moving out of its buffer. Since there is no provision made for backspacing, it would be more considerate to give the operator ten or twenty digits (in case of typing errors) and only examine the last five typed in. The easiest way to determine the last five bytes is to place a LD (HL),0 after the CALL as a terminator for the buffer. The following routine reads a buffer of five bytes and RETurns the value in HL:

```
;CONVERT ASCII DECIMAL INTO A NUMBER
;ENTRY -->    HL = START OF BUFFER


CONV    LD      IX,NTAB
        LD      HL,BUFF
        XOR     A               ;FIND END,
        LD      IY,0            ;OF BUFFER
        LD      BC,5            ;BUFFER LEN
        CPIR
        LD      A,5             ;CALCULATE,
        SUB     B               ;NUMBER OF,
        LD      C,A             ;DIGITS
        .   .   .
```

The actual string conversion is ready to begin:

```
LP1     DEC   HL          ;BACKWARDS
        LD    A,(HL)
        SUB   30H         ;TO ASCII
        JR    Z,LP2
        LD    B,A         ;LOOP
        LD    D,(IX+1)
        LD    E,(IX)
LP3     ADD   IY,DE
        JR    C,ERROR     ;OUT OF BOUNDS?
        DJNZ  LP3
LP2     INC   IX          ;SUBTRACT,
        INC   IX          ;NEXT
        DEC   C           ;COUNT DOWN
        JR    NZ,LP1
        PUSH  IY          ;PUT IY INTO,
        POP   HL          ;HL
        RET
ERROR   LD    HL,65535    ;IMPOSSIBLE TO,
        RET               ;CALCULATE

NTAB    DW    1,10,100,1000,10000
```

Initially the routine finds the buffer terminator using a block compare and calculates the number of times the main loop is to be executed. The register pair DE adds the appropriate power of 10 to IY, the number of times specified by B. If the number exceeds 65535 an overflow is produced and HL returns with 65535. The process repeats until all digits have been multiplied and added, ending with a final swap of the contents of HL and IY. In converting ASCII decimal to hexadecimal, we have subtracted and added base 10 numbers, but there is another approach I feel worth mentioning. This is multiplying and dividing by base 10. The following routine starts at the beginning of the ASCII buffer, reads the number, multiplies it by 10, gets the next number and adds it to the total, multiplies the total by 10 and so on. Examine the listing carefully:

```
;CONVERT ASCII DECIMAL TO NUMBER
;ENTRY -->     HL = START OF BUFFER
;              B  = NUMBER OF DIGITS

CONV    LD      IX,0
LOOP    PUSH    IX              ;SAVE IX
        ADD     IX,IX           ;TIMES 2
        ADD     IX,IX           ;TIMES 4
        POP     DE              ;RECOVER
        ADD     IX,DE           ;TIMES 5
        ADD     IX,IX           ;TIMES 10
        JR      C,OV            ;NUMBER TO BIG
        LD      A,(HL)          ;GET ASCII
        INC     HL              ;POINT TO NEXT
        LD      E,A             ;PUT IN DE
        LD      D,0
```

```
        ADD     IX,DE           ;TOTAL UP
        DJNZ    LOOP
        PUSH    IX
        POP     HL
        RET
OV      LD      HL,65535
        RET
```

Every time the number is multiplied by 10, all the digits are shifted one position (power of 10) to the left. Although it does less than the previous routine, it may be better suited to programs that have more sophisticated input decoders.

Now that the problem of transferring data is solved, our 'warm up' chapter is over. If you are unsure of any of the procedures discussed, review them carefully, in particular, study the movement of bits in each byte. The next chapters will move on to more sophisticated applications.

CHAPTER 4
– ORGANISING INFORMATION

Information has to be easily accessed, moved and manipulated, otherwise there is no point in storing data inside a computer in the first place. Routines are needed to search through lists, find pointers and display information in special ways. To make this possible, information has to be carefully organised in memory. This chapter will cover basic data handling techniques. More complex data structures are considered in chapter six.

4.1 ARRAYS

A *string* is a series of alpha-numeric characters s-t-r-u-n-g together with something in common. For example, each character in the word 'HARTNELL' is associated with every other one: they are all part of the one name.

A *list* or *table* has data ordered sequentially through memory, like information found in a shopping list. A *matrix* is a block of information organised in rows and columns. Of course it is impossible to represent data internally as anything but a list, but careful organisation will give it more complex characteristics. Matrices become especially useful when they are grouped together to form three dimensional arrays (used in generating cubic or three dimensional 'maps'). An *array* is the general name for either a list (one dimensional array), matrix (two dimensional array) or group of matrices (three dimensional array).

Arrays can have powers greater than three, depending on the application, but are less commonly used. A one or two dimensional array is often used for storing byte or word (two byte) pointers, and is called an *index*. A one dimensional array is referred to as; ARRAY(x). A two dimensional array as; ARRAY(x,y). A three dimensional array as: ARRAY(x,y,z).

Programmers familiar with high level languages will probably recognize array structures, and are aware of dimensioning (specifying the size of the array), and 'subscripted variables' (the x,y,z's). All these techniques deal with accessing information that have things in common.

The simplest application for a multi-dimensional array is a look up of information in tabulated form. We can simulate a table in memory by placing data in specific bytes. Using elementary mathematics variable ARRAY(x,y) can be calculated by:

$$\text{ARRAY} + \text{N} \times X + y$$

Where N is the length of the column, ARRAY is the base address of the table, x is the number of rows, and y the current column position. If we stored first just like a list, followed by the next and so on. Accessing the correct row is a matter of jumping a specified number of columns.

Diagram 4.0 represents a fictitious table of results for National Incomes and Expenditures (as would be used in an economics program). The columns represent years, the rows expenditure. Using a matrix we can access information by either. If we wanted to look at the first four items of expenditure in the 1982-83 period we would have to list variables ARRAY(2,0) to ARRAY(2,4).

Net Expenditure on Goods and Services	1980-81	1981-82	1982-83	1983-84	1984-85
Financial Enterprises	21	22	22	23	24
Public Authorities	102	95	98	101	105
Fixed Capital Equipment	87	89	98	110	115
Non-farm Stocks	25	60	40	61	12
Farm Stocks	5	15	9	21	26
Public Authorities	221	245	248	251	253

DIAGRAM 4.0 – TYPICAL INFORMATION TABLE

```
;TWO DIMENSIONAL ARRAY
;5 COLUMNS & 6 ROWS

        LD      BC,0002H    ;B=0, C=2 ARRAY
LOOP    CALL    ARRAY2
        CALL    PRINT       ;DISPLAY DATA
        INC     B
        CP      A,B
```

```
        CP      4               ;4 ITEMS
        RET     Z
        JR      LOOP            ;AGAIN
ARRAY2  PUSH    BC              ;KEEP BC
        EXX                     ;SAVE REGISTERS
        POP     BC              ;RECOVER
        LD      HL,ARRAY-5      ;TABLE,
        INC     B               ;MUST NOT BE 0
        LD      DE,5            ;ROW WIDTH
ALP     ADD     HL,DE           ;GET CORRECT,
        DJNZ    ALP             ;ROW
        ADD     HL,BC           ;TO COLUMN
        LD      A,(HL)          ;GET IT
        EXX
        RET
ARRAY   DB      21, 22, 22,23, 24
        DB      102,95, 98,101,105
        DB      87, 89, 98,110,115
        DB      25, 60, 40, 61, 12
        DB      5,  15, 9,  21, 26
        DB      221,245,248,251,253
```

The routine ARRAY2 calculates the row address by skipping in blocks of five bytes the number of times specified by B. The C register then points to the appropriate column, and the byte is loaded into HL. The routine CALLs ARRAY2, three times, RETurning the accumulator with 22, 98, 98 and 40 before exiting. To access a table of words, instead of bytes, the row length and column pointer would have to be doubled. An SLA will do this, provided that the number of elements in each subscript is not greater than 127.

Array manipulations are vital, and are an integral part of coordinate calculations (using an *x,y* axis), whether they are used to display information, examine the moves of an opponent in a computer chess game, determine the trajectory of a missile or alien in a video game, or whatever else must treat information two dimensionally. Matrix calculations are particularly important for games, and will be dealt with further in chapter five.

Groups of matrices can be used to manipulate three dimensional data, although this application is relatively rare. Rather, ARRAY(*x,y,z*) is normally used to point to sets of tables, with *'z'* as the 'page' or 'book' number.

4.2 MOVING DATA

Data has to be relocated and shifted, and giving full credit to the Z80's sophisticated block moving capability, this can be done with a minimum of coding. Yet moving data, *the way you want to,* is not necessarily a straightforward procedure. Overlap may overwrite a section of code before it is fully moved, a miscalculated block shift simply fill memory.

Block move parameters are logical enough:

```
MOVE    LD      HL,5000     ;SOURCE
        LD      DE,7000     ;DESTINATION
        LD      BC,1023     ;1K - 1 BYTE
        LDIR
```

Using the same technique we could fill a portion of memory by:

```
MOVE1   LD      HL,5000     ;SOURCE
        LD      DE,5001     ;DESTINATION
```

```
        LD      BC,1023     ;1K - 1 BYTE
        LD      (HL),A      ;BYTE IN A
        LDIR
```

This is an example of a shift that loads 1K of data starting at 5000 with the value in the accumulator. This could be used to clear the VDU for example, or zero some section of memory. When designing routines to shift or move data, consider each instruction as a set of small load and increment instructions. Know when instructions get loaded and incremented. For example, the LDIR first loads the contents of HL into DE. The address pointed to by DE now holds the value in the accumulator. The register pair HL is incremented to point to DE and DE is incremented to point to the next byte. Then BC is decremented, and the process repeats until BC is zero.

All this is invisible since LDIR (repeating block load with increment) is an invisible self-contained subset of instructions. This type of block move is called a destructive one, since all that it does is carry the byte starting at 5000 along 1K regardless of other memory. What is more useful is a non-destructive shift that moves a block of data a byte up or down in memory. We could shift a block of data one byte to the left (to a lower memory address) by executing the following routine:

```
MOVE2   LD      HL,5001     ;SOURCE
        LD      DE,5000     ;DESTINATION
        LD      BC,1023     ;SHIFT 1K
        LDIR
```

Carefully examine this block move as well. The contents of 5001 are moved into 5000 *before* the contents of 5001 are changed. On the second re-execution of LDIR, HL points to 5002 and DE to 5001. Now that the contents of 5001 have been moved to 5000, it is okay to move the contents of 5002 to 5001. No data is lost in the process.

If we wanted to do the opposite, that is, shift a block of data one byte to the right (to a higher memory address), we have to approach the problem from a different angle:

```
MOVE3   LD      HL,5998     ;SOURCE
        LD      DE,5999     ;DESTINATION
        LD      BC,1023     ;SHIFT 1K - 1
        LDDR
```

The LDDR instruction (repeating block load with decrement) is identical to LDIR except DE and HL are decremented instead of incremented. In order not to destroy data it works backwards this time. Register pair HL points to the end of the block to move minus one. Register pair DE points to the last byte, so the contents of HL can be moved to DE, and saved away. The principle is then identical to that shown in MOVE2.

Block move instructions can be troublesome, as already discussed in chapter one, but it is important to re-emphasize this here. Be careful not to lose the top or bottom byte when shifting. Be very careful that BC has the correct value. It is easy to accidentally move one byte too many or too few.

By changing the destination address, bytes can be shifted in sets of blocks rather than single bytes. This is especially useful where there is a need to insert or delete information from a table (for example a word processor). The following routine will scroll the display up one row on your VDU:

```
MOVE4   LD      HL,VDU+COL  ;TOP + 1 COLUMN
        LD      DE,VDU      ;TOP OF VDU
        LD      BC,PNT-VDU  ;END OF,
        LDIR                ;VDU-COLUMN
```

The label VDU is the start of VDU RAM, COL is the column length, and PNT is the bottom left hand corner of the VDU. This is calculated by subtracting the very last VDU location from the column length *and adding one.* To scroll down:

```
MOVE5   LD      HL,PNT-COL  ;BOTTOM - 1 ROW
        LD      DE,PNT      ;BOTTOM LEFT
        LD      BC,PNT-VDU
```

```
        LDDR
```

If you want to avoid the problem of overlap, without worrying too much about the direction of the move, the last subroutine will automatically make the necessary calculations for you. On entry HL, DE and BC must be loaded with the usual source, destination and bytes to move.

```
MOVE6   OR      A
        SBC     HL,DE           ;LDDR OR LDIR?
        JR      C,PASS          ;OVERFLOW
        ADD     HL,DE           ;OLD VALUE
        LDIR
        RET
PASS    ADD     HL,DE           ;OLD VALUE
        DEC     BC              ;ADJUST BC
        ADD     HL,BC           ;END OF BLOCK
        EX      DE,HL
        ADD     HL,BC           ;END OF BLOCK
        EX      DE,HL
        INC     BC              ;ADJUST BC
        LDDR                    ;BACKWARDS
        RET
```

4.3 STRING MANIPULATIONS

Programmers need to work with strings to find fields, decode instructions and display messages. This section is a summary of useful string manipulation subroutines.

The most straightforward string manipulation is a string compare:

```
;STRING COMPARE
;ENTRY -->    DE = STRING 1
;             HL = STRING 2
;EXIT  -->    Z  = 1 IF EQUAL

STRCP  LD     A,(DE)        ;STRING 1
       CP     (HL)          ;STRING 2
       INC    HL
       INC    DE
       JR     Z,STRCP
       INC    A
       RET                  ;Z=1 IF MATCHED

STR1   DB     'THE GOLDEN TORC',255
STR2   DB     'THE GOLDEN TORC',255
```

Which compares two strings byte for byte. On exit, the Z flag is set if both are equal. Both STR1 and STR2 are sample strings, requiring a 255 byte terminator.

String *concatenation* combines two strings together. This is a matter of locating the end of the first string and then moving the second string into position behind it. The HL register points to the start of the first string, which must have a 255 byte terminator and DE to the second, also with a 255 byte terminator. The end of the first string is automatically calculated in the program:

```
;STRING CONCATENATION

;ENTRY -->    HL = MAIN STRING

;             DE = STRING TO APPEND


STRCO   LD    A,255          ;FIND END OF,
        LD    BC,-1          ;FIRST STRING
        CPIR
        DEC   HL             ;HACK LAST BYTE
        EX    DE,HL          ;SWAP
GOTIT   LD    A,(HL)         ;REACHED END?
        LDI                  ;MOVE
        INC   A              ;NEW VALUE Ø?
        JR    NZ,GOTIT
        DEC   A
        LD    (HL),A         ;TERMINATOR = Ø
        RET
```

As a precaution all the bits in the BC register pair are set to make sure a match is found before BC terminates execution of the CPIR instruction. Be careful that you have enough room at the end of the first string to allow concatenation of the second.

Finding the substring of a string is an even more useful application. The following is a substring search routine.

```
;RETURN OFFSET OF SUBSTRING IN A STRING
;ENTRY -->    DE = MAIN STRING
;             HL = SUBSTRING
;EXIT  -->    BC = OFFSET (STRING POS)
;             CARRY = 1, NO MATCH
;             ELSE FOUND

FSUB    LD    (ADDR1),HL
FSUB1   LD    BC,1          ;ADD OFFSET
        LD    HL,(ADDR1)
FSUB2   LD    A,(DE)
        INC   A             ;END OF STRING?
        SCF                 ;NOT FOUND YET
        RET   Z
        DEC   A             ;RECOVER
        CP    (HL)          ;COMPARE TO,
        INC   HL            ;SUBSTRING
        INC   DE
        INC   BC
        JR    Z,FSUB2       ;SO FAR GOOD
        LD    A,(HL)        ;CHECK END
```

```
        INC     A               ;IS IT 255?
        JR      NZ,FSUB1        ;NO
        OR      A               ;CLEAR CARRY
        RET
ADDR1   DW      0
STR1    DB      'COUNCIL OF ELROND',255
STR2    DB      'OF',255
```

The BC register pair returns with an offset pointing to the start of the substring. Using this routine, substrings can be inserted or deleted using block shift routines.

4.4 DATA COMPRESSION

Since the Z80 is limited to 65536 addressable memory locations just about every application is memory critical. When consumptious amounts of RAM are used by ROM or VDU memory, available 'free space' is narrowed even further. Few Z80 programs exceed 16K, however, when large blocks of memory are reserved for VDU images, whether they picture a scene from a game or a graphical representation of some other information, memory can run out very quickly. In a bid to save as much memory as possible we have to look at what wastes the most bytes in such images. These are usually the blank spaces between words and sentences or the background of a picture.

By replacing these blanks with a number code representing the bytes to 'skip' it is possible to save perhaps 80 per cent of memory and in some cases even more. The same method would also enable the programmer to superimpose one image over another.

Listing 4.0 is a graphic code packing program designed around the TRS-80, although it will work on any computer. Colour computers must check for the background colour, monochrome computers need only change the VDU RAM addresses. On this particular machine however, bytes 128-191 are reserved for graphic blocks. A 128 is a blank space, and 191 is an all white block, with all combinations in between. Lower codes are used for ASCII and the upper for special symbols. Since none of these enter the picture we can use them for other purposes.

The routine PACK1 starts by checking for a blank space. If it isn't, 64 is added to the graphic code, giving it a new value between 192-255 which is stored away in the buffer. Otherwise the routine SKIP counts the number of blanks and then stores this away in the buffer as well. If the number looks like it's going to be larger than 192 spaces it stores the current number in the buffer and begins again. When finished, the buffer (pointed to by IX) contains all graphics as numbers above 191 and everything below as counters for spaces to skip. Listing 4.1 decodes the compressed format, returning the graphics to their original value by subtracting 64, before displaying.

```
                00100 ;GRAPHIC CODE PACKER
                00110 ;ENTRY -->      IX = DESTINATION
                00120 ;               HL = SOURCE
                00130 ;               DE = AMOUNT
                00140 ;NOTES: BYTES 128-191 ONLY (GRAPHIC CODES)
                00150 ;*ROUTINE MUST BE MODIFIED TO SUIT CHARACTER SET
                00160 ;*********************************************
                00170 ;
9000            00180          ORG     9000H            ;*
9000 CD0890     00190 PACK     CALL    PACK1            ;DATA PACK
9003 DD360000   00200          LD      (IX),0           ;TERMINATOR
9007 C9         00210          RET                      ;EXIT
                00220 ;
9008 7E         00230 PACK1    LD      A,(HL)
9009 FE80       00240          CP      128
900B 280E       00250          JR      Z,SKIP           ;ZAP BLANK
900D C640       00260          ADD     A,64             ;MOVE TO 192-255
900F DD7700     00270          LD      (IX),A           ;SAVE IT
9012 23         00280          INC     HL
9013 DD23       00290 PACK2    INC     IX
9015 1B         00300          DEC     DE
9016 7A         00310          LD      A,D
9017 B3         00320          OR      E
9018 C8         00330          RET     Z                ;FINISHED?
9019 18ED       00340          JR      PACK1
901B 0600       00350 SKIP     LD      B,0              ;COUNT BLANKS
901D 7E         00360 SKIP1    LD      A,(HL)
901E FE80       00370          CP      128              ;MORE BLANKS?
9020 200E       00380          JR      NZ,COM
9022 04         00390          INC     B
9023 78         00400          LD      A,B              ;CHECK IF BLANKS EXCEED,
9024 FEBE       00410          CP      190              ;NUMERIC LIMIT
9026 CC3990     00420          CALL    Z,STORE
9029 1B         00430          DEC     DE
902A 7A         00440          LD      A,D
902B B3         00450          OR      E
902C C8         00460          RET     Z
902D 23         00470          INC     HL
902E 18ED       00480          JR      SKIP1
9030 78         00490 COM      LD      A,B
9031 B7         00500          OR      A
9032 28D4       00510          JR      Z,PACK1
9034 DD7000     00520          LD      (IX),B           ;STORE NO. OF BLANKS
9037 18DA       00530          JR      PACK2
9039 DD7000     00540 STORE    LD      (IX),B           ;STORE NO. OF BLANKS
903C DD23       00550          INC     IX
903E 0600       00560          LD      B,0
9040 C9         00570          RET
9000            00580          END     PACK
00000 Total errors
```

```
                00100 ;GRAPHIC CODE UNPACKER
                00110 ;ENTRY -->        HL = SOURCE
                00120 ;                 DE = AREA TO DISPLAY
                00130 ;*ROUTINE MUST BE MODIFIED TO SUIT CHARACTER SET
                00140 ;*********************************************
                00150 ;
9000            00160           ORG     9000H           ;*
9000 7E         00170 UNPACK    LD      A,(HL)
9001 B7         00180           OR      A               ;DONE?
9002 C8         00190           RET     Z
9003 FEC1       00200           CP      193             ;IS IT GRAPHIC?
9005 3007       00210           JR      NC,SHOW
9007 47         00220           LD      B,A             ;POSITIONS TO SKIP
9008 13         00230 LOOP      INC     DE
9009 10FD       00240           DJNZ    LOOP
900B 23         00250 SHOW1     INC     HL
900C 18F2       00260           JR      UNPACK
900E D640       00270 SHOW      SUB     64
9010 12         00280           LD      (DE),A          ;DISPLAY IT!
9011 13         00290           INC     DE              ;NEXT BYTE
9012 18F7       00300           JR      SHOW1
9000            00310           END     UNPACK
00000 Total errors
```

Although this routine is fairly specialized in many respects (the actual character set layout) it does demonstrate how data can be compressed. Even programs that are not memory critical will benefit since the program occupies less space on a mass storage device and is faster to load in.

ASCII, a 7 bit code, can also be slightly compacted. The last bits of every 7 bytes can be used to create another byte. This adds up to almost 15 per cent savings. When writing data compression routines keep in mind how many bits it takes to hold what you need. Alpha-numerics; the alphabet plus numerals 0-9 need never exceed 36. If we examined bits 6 and 7 we would note that they were both reset at 36. These could be used to hold more alpha-numerics or other data.

4.5 COMMAND TABLES

A command table is used to 'look up' the address of a routine and transfer control (jump) to it. The editor/assembler program for example has a number of one letter commands including I(nsert), D(elete), and E(dit) that must pass control to special subroutines. One method is to compare for each command:

```
        CALL   GKEY
        CP     'E'
        JP     Z,EDIT
        CP     'I'
        JP     Z,INSERT
        CP     'D'
        JP     Z,DELETE
```

But this clumsy and cumbersome method soon gets out of hand as more instructions are introduced. The solution is to create two tables; one holding the ASCII characters, the other a set of addresses. A generalized compare routine then matches the address to the character. First we must get the command, specify how many instructions exist, and then search through a list of possibilities:

```
COMM    CALL   GKEY          ;GET CHARACTER
        LD     BC,10         ;COMMAND COUNT
        LD     HL,CTABLE     ;POINT TO TABLE
        LD     DE,0FFFFH     ;COUNTER
AGAIN   CPI
        INC    DE
        JP     PO,COMM     ;NO COMMAND!
        JR     NZ,AGAIN
        .  .  .
```

The CALL GKEY places a character in the accumulator, BC specifies the length of the table. The register pair HL is loaded with the base address of the table, and CPI compares the contents of HL to A. If BC is zero, end of the table has been reached, parity is reset, and no match is found, so, it tries again. If the accumulator finds a match, the program follows through:

```
LD      IX,VECTOR   ;JUMPS
SLA     E           ;MULTIPLY BY 2
ADD     IX,DE       ;POINT VECTOR
LD      L,(IX)      ;GET JUMP,
LD      H,(IX+1)    ;ADDRESS
JP      (HL)        ;EXECUTE
```

Register IX is loaded with the vector table containing the memory location of each routine. The routine must find which address to jump to. The E register is doubled (because an address is two bytes long) to align it with the corresponding VECTOR address in the second table. It then adds the offset, in DE, to IX, and points PC to the correct address. The data tables would look something like the following:

```
CTAB    DB      'EIDCPKOLW'
VECTOR  DW      EDIT,INSERT,DELETE,COPY
        DW      PRINT,KILL,OPEN,LIST,WRITE
```

This type of command table is acceptable for applications where the operator is expected to have experience. Monitor programs and editor/assemblers (or where all information is displayed in menu form) are programs that fit this description. There are times however when a menu is unsuitable, or, the VDU is occupied displaying other information. In business, and many other applications, one letter commands (that inevitably have to be memorised) are unacceptable. English instruction is one alternative.

Writing a program that understands English is challenging and fruitful. Provided that you have taken time and effort, utilised arrays, data packing, and especially string manipulation techniques, any program will be able to decode English words or complete English sentences. Programming languages like BASIC use a debased form of English to manipulate the computer's functions. Database management

tools often require English words for instruction, like 'BUILD A FILE' and 'FIND ADAM'. Artificial intelligence is another possibility, well publicized by the 'ELIZA' program.

Furthermore, it's often applied in 'Adventure' games. The operator communicates in English, or a simplified form of it, to interact with the story line. Written mainly for entertainment, they can predominantly be complex puzzles or small adventure stories, where the operator is given the option of changing its outcome by making decisions at critical stages throughout the program. Unfortunately they are rarely used as tests, requiring far more thought and initiative than anything a pen and paper examination could offer.

Study a typical sentence to be decoded by a computer:

GET EVERYTHING OFF THE TABLE EXCEPT THE MATCHES AND THE PIN.

At first glance it may seem like an impossible task. Storing away entire sentences like this, and then string comparing is impractical, since it just as easily could have been rewritten as:

GRAB ALL THE ITEMS ON THE TABLE BUT LEAVE THE PIN AND MATCHES.

Which retains the same meaning in a different form. The sentence has to be broken down into comprehensible units, using substring searching, and deleting. The first stage is to ignore or remove superfluous adjectives:

GET EVERYTHING *OFF THE* TABLE EXCEPT *THE* MATCHES *AND THE* PIN.

Words like 'THE' have grammatical importance in English but do not add to the idea of the sentence. The general meaning is still retained if we wrote:

GET EVERYTHING – TABLE EXCEPT – MATCHES – PIN.

The word 'EVERYTHING' tells us that the operator wants everything on the table, not the table itself, and there is no need for an 'AND' between 'MATCHES' and 'PIN'. A comma could have the same meaning. We are mainly interested in two parts of the sentence; nouns, to identify what must be manipulated; and verbs, to tell us what to do with them. The first word of a sentence of this structure will always be a verb. By searching through a series of verbs stored away in a table, we come to the 'GET' routine, and jump to it. We now have left:

EVERYTHING TABLE EXCEPT MATCHES PIN.

The word 'EVERYTHING' tells us to get all possible items. The GET routine searches through another table checking which objects can be 'picked up' and adjusts data appropriately. Words like 'PIN' and 'MATCH' must be converted, by use of a table, into their numeric equivalents (the program assigns a number to each word). Leaving us with:

EXCEPT MATCHES PIN.

Finally the GET routine must search through a preposition table containing words like 'EXCEPT', 'FROM', 'WITH', etc. to identify 'EXCEPT', and execute a 'DROP MATCHES' and 'DROP PIN' command. Depending on where a word is in a sentence, prepositions could simply be ignored, like adjectives, or decoded, like the word 'EXCEPT'. It may seem like a lot of work, since there are many ways a sentence can be written, and in fact it is. You must take care in considering all the reasonable sentence structures, not just one or two. The point is that English can be broken down into something computer comprehensible, as long as we reduce it to a string of numbers.

Demonstrating such a large decoding routine distracts from the principle of its operation, so for the sake of clarity I have settled for something smaller: a two-word English decoder. The first must be a verb, the second a noun (you could change this to suit yourself, but short sentences like GET ROPE, PEEL BANANA, OPEN DOOR in the verb/noun format can be used to express just about every wish). Writing a suitable input routine is no problem. Once the command has been received it must be decoded. To do this, the operator's word must be compared with a list of words in memory. Listing 4.2, a word decoder routine, does the comparing and matching.

It begins by loading IY with a VECTOR table, holding the starting address of each execution routine (the routine that performs the GET function in a GET command etc.). The index register IX is loaded with the words the computer understands, and HL points to the word the operator typed in. Line 200 begins with a character compare. When unequal, another compare is made to check if IX points to the word terminator (the symbol #). If it does, the word is equal enough, and a jump is made to the address held in IY. If not, IX is incremented until it points to the next word, IY is incremented by two, to point to the next address, and the procedure repeats. A 255 byte is an end of table indicator. It means all possible entries have been checked and the routine would RETurn, where a message like 'SORRY I DON'T KNOW THAT WORD' or something to that effect would be displayed.

```
                00100 ;COMPARE STRING
                00110 ;
9000            00120           ORG     9000H           ;*
9000 FD214A90   00130 SEARCH    LD      IY,VECTOR       ;COMMAND ADDRESS
9004 DD213C90   00140           LD      IX,TABLE        ;STRINGS
9008 213B90     00150 SLOOP     LD      HL,COM          ;COMMAND TO MATCH
900B 7E         00160 REPEAT    LD      A,(HL)          ;GET BYTE TO COMPARE
900C 23         00170           INC     HL              ;POINT TO NEXT BYTE
900D DD4600     00180           LD      B,(IX)          ;GET BYTE OF OTHER STRING
9010 DD23       00190           INC     IX
9012 B8         00200           CP      B               ;MATCHED?
9013 28F6       00210           JR      Z,REPEAT        ;SO FAR...
9015 78         00220           LD      A,B             ;COMPARE IF STRING MATCHED
9016 FE23       00230           CP      '#'             ;IS IT?
9018 2817       00240           JR      Z,FOUND         ;YES!
901A FEFF       00250           CP      255             ;END OF LIST?
901C 2813       00260           JR      Z,FOUND
901E DD23       00270 AGAIN     INC     IX              ;POINT TO NEXT STRING
9020 DD7E00     00280           LD      A,(IX)
9023 3C         00290           INC     A               ;IS IT 255?
9024 C8         00300           RET     Z               ;END OF LIST, NO MATCH
9025 FE24       00310           CP      '#'+1           ;GOT NEXT WORD?
9027 20F5       00320           JR      NZ,AGAIN        ;NOT YET?
9029 DD23       00330           INC     IX              ;FIRST LETTER
902B FD23       00340           INC     IY              ;NEXT ADDRESS
902D FD23       00350           INC     IY
902F 18D7       00360           JR      SLOOP           ;AND TRY AGAIN
9031 FD6601     00370 FOUND     LD      H,(IY+1)        ;JUMP TO ADDRESS
9034 FD6E00     00380           LD      L,(IY)
9037 E9         00390           JP      (HL)            ;GO THERE!
                00400 ;
9038 47         00410 COM       DB      'GET',0         ;OPERATOR COMMAND
     45 54 00
903C 52         00420 TABLE     DB      'RUN#LOOK#GET#',255
     55 4E 23 4C 4F 4F 4B 23
     47 45 54 23 FF
904A 5290       00430 VECTOR    DW      RUN,LOOK,GET
     5190 5090
9050 00         00440 GET       NOP                     ;ROUTINES HERE:
9051 00         00450 LOOK      NOP
9052 00         00460 RUN       NOP
9000            00470           END     SEARCH
00000 Total errors
```

What exactly do I mean by *equal enough*? If you typed GETHDSHD, instead of GET, it would still be a valid input, since only the first three letters are required. A check could be made for 'GET_' to make sure there is a space after the word, but this is carrying comparing to the degree of paranoia. Abbreviated words are quicker and easier (if there is a very slight chance of being misinterpreted) to type in. Words like GET and LOOK require you to spell them out completely since they are only made up of a few letters, but it's not very common for adventures to check more than the first five letters of a long word. Having to type 'GET PHOSPHORUS' twenty times an hour is not amusing.

You will notice that each word in TABLE is separated by a # symbol, which is used as a divider for RUN, LOOK and GET. This is necessary since the length of each word is inconsistent, and we have to find the beginning of the next one. An alternate, and much better way is to place a byte specifying the length of the word (from 1-5) before the actual word. This discards the need for having the AGAIN

routine in line 270. Both methods nevertheless waste a byte for a divider or counter. Setting bit 7 (bit 7 is always reset in ASCII) of the first letter in each word doesn't waste an extra byte, however, both methods take up more editor/assembler buffer space, are less readable and take longer to type in. So, writing to make the job easier for the programmer, I have divided each word with conventional ASCII. I recommend you do the same when main memory is not as critical as your editor/assembler.

If you wanted to process small sentences, routines could be written to interpret only the first and last words in a sentence. This would allow operators to type: GET THE ROPE, PEEL THE BANANA, and OPEN THE DOOR. Beyond that, more complex methods must be tried.

The techniques discussed would be unsuitable for English beyond three words. After which it becomes necessary to 'chop up' sentences using substring deleting. They must then be decoded into strings of numbers, and these number 'patterns' processed. The machine language is essentially not too difficult, but a lot of thought must be put into which sentences you intend to decode. The degree of English your program will understand will depend solely on the amount of effort you are willing to put into the project.

4.6 ADVENTURE GAMES

Examine listing 4.3, a mini-adventure that uses the SEARCH routine discussed in the last section. The program prepares the VDU, displays the name of the location, and then waits for keyboard input. It accepts all characters (using a ROM input routine – to be changed to suit your ROM, or possibly CALL your own keyboard decoding routine) from the keyboard. A code '8' is an ASCII 'backspace' which moves back over text to correct syntax, and a code '13' is a 'return' to be pressed when the sentence is completed.

```
                00100 ;ADVENTURE GAME/ENGLISH WORD DECODER
                00110 ;NOTE: 1 OR 2 WORD DECODER
                00120 ;       C = 0 IF ONLY 1 WORD IS USED BEFORE JP (HL)
                00130 ;       OR WHEN 2ND WORD IS UNKNOWN
                00140 ;*WORKS WITH STANDARD ASCII CHARACTERS
                00150 ;*CHANGE ACCORDING TO COMPUTER
                00160 ;*****************************
                00170 ;
3C00            00180 VDU       EQU     15360           ;*START OF VDU RAM
03FF            00190 LEN       EQU     1023            ;*LENGTH OF VDU RAM-1
3FC0            00200 BTL       EQU     15360+960       ;*BOTTOM LEFT-HAND CORNER
0040            00210 COLUMN    EQU     64              ;*COLUMN LENGTH
0049            00220 KBROMC    EQU     49H             ;*KEYBOARD ROM CALL
9000            00230           ORG     9000H           ;*
9000 F3         00240 ADVENT    DI
9001 31FFFF     00250           LD      SP,65535        ;*
9004 CD3191     00260           CALL    INIT            ;ORGANISE RAM
9007 CDE690     00270 ALP       CALL    LOCAT           ;EXECUTION CONTROL LOOP
900A 21A091     00280           LD      HL,BUFF         ;CLEAR LAST,
900D 11A191     00290           LD      DE,BUFF+1       ;WORD OUT OF,
9010 012700     00300           LD      BC,39           ;BUFFER
9013 3600       00310           LD      (HL),0
9015 EDB0       00320           LDIR
9017 CD2890     00330           CALL    INPUT           ;GET WORD
901A CD1091     00340           CALL    SCROLL          ;MOVE SCREEN UP
901D CD5F90     00350           CALL    DECODE          ;EXECUTE INSTRUCTIONS
9020 214391     00360           LD      HL,NO           ;"DON'T KNOW THAT WORD"
9023 CD0491     00370           CALL    PRINT
9026 18DF       00380           JR      ALP
                00390 ;
9028 21A091     00400 INPUT     LD      HL,BUFF         ;WORD BUFFER
902B 11C03F     00410           LD      DE,BTL          ;ON VDU
902E 0600       00420           LD      B,0             ;COUNTER
9030 3E5F       00430 INPUT1    LD      A,95            ;ASCII CURSOR
9032 12         00440           LD      (DE),A          ;DISPAY IT
9033 CD2B91     00450           CALL    GKEY            ;GET CHARACTER
9036 FE08       00460           CP      8               ;IS IT A BKSP?
9038 2818       00470           JR      Z,BKSP
903A FE0D       00480           CP      13              ;IS IT A CR?
903C 280E       00490           JR      Z,RETURN
903E F5         00500           PUSH    AF
903F 78         00510           LD      A,B
9040 FE28       00520           CP      40              ;BUFFER FULL?
9042 F1         00530           POP     AF
9043 28EB       00540           JR      Z,INPUT1        ;YES, BACK?
9045 04         00550           INC     B
9046 77         00560           LD      (HL),A          ;STORE IN BUFFER
9047 12         00570           LD      (DE),A          ;DISPLAY ON VDU
9048 13         00580           INC     DE              ;ADJUST POINTERS
9049 23         00590           INC     HL
904A 18E4       00600           JR      INPUT1
904C 3600       00610 RETURN    LD      (HL),0          ;TERMINATOR BYTE
904E 3E20       00620           LD      A,32            ;CLEAR CURSOR
```

```
9050 12          00630          LD      (DE),A
9051 C9          00640          RET
9052 78          00650 BKSP     LD      A,B             ;CHECK IF BUFFER,
9053 B7          00660          OR      A               ;EMPTY
9054 28DA        00670          JR      Z,INPUT1        ;YES?
9056 05          00680          DEC     B               ;REMOVE LAST CHARACTER
9057 2B          00690          DEC     HL
9058 3E20        00700          LD      A,32            ;CLEAR OFF VDU
905A 77          00710          LD      (HL),A
905B 12          00720          LD      (DE),A          ;REMOVE FROM BUFFER
905C 1B          00730          DEC     DE              ;BACK
905D 18D1        00740          JR      INPUT1
                 00750 ;
905F FD218C91    00760 DECODE   LD      IY,VECTOR       ;JUMP ADDRESS
9063 DD218E91    00770          LD      IX,TABLE        ;TABLE OF VERBS
9067 21A091      00780 S1       LD      HL,BUFF         ;THE COMMAND
906A 7E          00790 REPEAT   LD      A,(HL)
906B 23          00800          INC     HL
906C DD4600      00810          LD      B,(IX)
906F DD23        00820          INC     IX
9071 B8          00830          CP      B               ;ARE LETTERS,
9072 28F6        00840          JR      Z,REPEAT        ;THE SAME?
9074 78          00850          LD      A,B             ;NO
9075 FE23        00860          CP      '#'             ;REACHED END,
9077 2816        00870          JR      Z,FOUND         ;OF WORD?
9079 3C          00880          INC     A               ;A=255
907A 2813        00890          JR      Z,FOUND
907C DD23        00900 AGAIN    INC     IX              ;SKIP
907E DD7E00      00910          LD      A,(IX)
9081 3C          00920          INC     A
9082 C8          00930          RET     Z               ;END OF TABLE?
9083 FE24        00940          CP      '#'+1           ;POINT TO NEW,
9085 20F5        00950          JR      NZ,AGAIN        ;WORD
9087 DD23        00960          INC     IX
9089 FD23        00970          INC     IY              ;POINT TO NEW,
908B FD23        00980          INC     IY              ;ADDRESS
908D 18D8        00990          JR      S1
908F FD229B91    01000 FOUND    LD      (ADDR1),IY      ;GOT 1ST WORD
9093 2B          01010          DEC     HL
9094 7E          01020 FLP      LD      A,(HL)          ;SKIP REST OF WORD
9095 B7          01030          OR      A
9096 2840        01040          JR      Z,NOWORD        ;IF ZERO, NO 2ND WORD
9098 FE20        01050          CP      32
909A 2803        01060          JR      Z,NEXT
909C 23          01070          INC     HL
909D 18F5        01080          JR      FLP
909F 7E          01090 NEXT     LD      A,(HL)          ;POINT TO 2ND WORD
90A0 B7          01100          OR      A
90A1 2835        01110          JR      Z,NOWORD        ;NO 2ND WORD?
90A3 FE20        01120          CP      32
90A5 2003        01130          JR      NZ,NEXT1
90A7 23          01140          INC     HL
90A8 18F5        01150          JR      NEXT
90AA 0E01        01160 NEXT1    LD      C,1             ;C=1ST WORD
90AC DD219391    01170          LD      IX,TABLE2       ;TABLE OF NOUNS
90B0 229D91      01180          LD      (ADDR2),HL
90B3 2A9D91      01190 S2       LD      HL,(ADDR2)
90B6 7E          01200 RLP      LD      A,(HL)
90B7 23          01210          INC     HL
90B8 DD4600      01220          LD      B,(IX)
90BB DD23        01230          INC     IX
90BD B8          01240          CP      B
90BE 28F6        01250          JR      Z,RLP
90C0 78          01260          LD      A,B
90C1 FE23        01270          CP      '#'
90C3 2815        01280          JR      Z,GOTIT         ;GOT 2ND WORD
90C5 3C          01290          INC     A
90C6 2812        01300          JR      Z,GOTIT         ;GOT 2ND WORD
90C8 DD23        01310 AGAIN2   INC     IX
90CA DD7E00      01320          LD      A,(IX)
```

```
90CD 3C       01330          INC     A
90CE C8       01340          RET     Z
90CF FE24     01350          CP      '#'+1
90D1 20F5     01360          JR      NZ,AGAIN2
90D3 DD23     01370          INC     IX
90D5 0C       01380          INC     C
90D6 18DB     01390          JR      S2
90D8 0E00     01400 NOWORD   LD      C,0             ;NO 2ND WORD, C=0
90DA FD2A9B91 01410 GOTIT    LD      IY,(ADDR1)      ;JUMP TO ADDRESS
90DE FD6E00   01420          LD      L,(IY)          ;HL=ADDRESS
90E1 FD6601   01430          LD      H,(IY+1)
90E4 79       01440          LD      A,C             ;C=CODE FOR NOUN
90E5 E9       01450          JP      (HL)            ;GO TO CORRECT WORD
              01460 ;
90E6 3A9F91   01470 LOCAT    LD      A,(ROOM)        ;1 DIMENSIONAL ARRAY
90E9 87       01480          ADD     A,A             ;DOUBLE TO GET ADDRESS
90EA 219991   01490          LD      HL,TABLE3       ;TABLE OF MESSAGES
90ED 5F       01500          LD      E,A
90EE 1600     01510          LD      D,0
90F0 19       01520          ADD     HL,DE           ;POINT TO IT
90F1 5E       01530          LD      E,(HL)
90F2 23       01540          INC     HL
90F3 56       01550          LD      D,(HL)
90F4 EB       01560          EX      DE,HL           ;HL=LOCATION
90F5 C30491   01570          JP      PRINT           ;DISPLAY IT
              01580 ;
90F8 FE01     01590 LOOK     CP      1               ;SAMPLE COMMAND
90FA C0       01600          RET     NZ              ;ROUTINE
90FB 215E91   01610          LD      HL,M1           ;IF C=1 (WINDOW)
90FE CD0491   01620          CALL    PRINT           ;SAY "TOO DIRTY"
9101 C30790   01630          JP      ALP             ;RETURN
              01640 ;
              01650 ;SUBROUTINE PACKAGE:
9104 11C03F   01660 PRINT    LD      DE,BTL          ;DISPLAY MESSAGE
9107 7E       01670 PLP      LD      A,(HL)
9108 B7       01680          OR      A
9109 2805     01690          JR      Z,SCROLL        ;SCROLL VDU?
910B 12       01700          LD      (DE),A
910C 13       01710          INC     DE
910D 23       01720          INC     HL
910E 18F7     01730          JR      PLP
              01740 ;
9110 D9       01750 SCROLL   EXX                     ;SAVE GENERAL REGISTERS
9111 21403C   01760          LD      HL,VDU+COLUMN   ;MOVE UP
9114 11003C   01770          LD      DE,VDU
9117 01C003   01780          LD      BC,BTL-VDU
911A EDB0     01790          LDIR
911C 21C03F   01800          LD      HL,BTL          ;CLEAR BOTTOM COLUMN
911F 11C13F   01810          LD      DE,BTL+1
9122 013F00   01820          LD      BC,COLUMN-1
9125 3620     01830          LD      (HL),32
9127 EDB0     01840          LDIR
9129 D9       01850          EXX
912A C9       01860          RET
              01870 ;
912B D5       01880 GKEY     PUSH    DE              ;*
912C CD4900   01890          CALL    KBROMC          ;*ROM CALL OR DECODER HERE
912F D1       01900          POP     DE              ;*
9130 C9       01910          RET
              01920 ;
9131 21003C   01930 INIT     LD      HL,VDU          ;CLEAR VDU
9134 11013C   01940          LD      DE,VDU+1
9137 3620     01950          LD      (HL),32
9139 01FF03   01960          LD      BC,LEN
913C EDB0     01970          LDIR
913E AF       01980          XOR     A               ;1ST LOCATION
913F 329F91   01990          LD      (ROOM),A
9142 C9       02000          RET
              02010 ;
```

```
9143 53        02020 NO       DB       'SORRY, I DO NOT UNDERSTAND',0
     4F 52 52 59 2C 20 49 20
     44 4F 20 4E 4F 54 20 55
     4E 44 45 52 53 54 41 4E
     44 00
915E 53        02030 M1       DB       'SORRY,     DIRTY.',0
     4F 52 52 59 2C 20 54 4F
     20 44 49 52 54 59 2E 00
916F 49        02040 R1       DB       'I AM IN A DEN WITH A WINDOW.',0
     20 41 4D 20 49 4E 20 41
     20 44 45 4E 20 57 49 54
     48 20 41 20 57 49 4E 44
     4F 57 2E 00
918C F890      02050 VECTOR   DW       LOOK
918E 4C        02060 TABLE    DB       'LOOK',255
     4F 4F 4B FF
9193 57        02070 TABLE2   DB       'WINDO',255
     49 4E 44 4F FF
9199 6F91      02080 TABLE3   DW       R1
               02090 ;VARIABLES:
919B 0000      02100 ADDR1    DW       0              ;ADDRESS OF WORD ROUTINE
919D 0000      02110 ADDR2    DW       0              ;NOUNS STORAGE
919F 00        02120 ROOM     DB       0              ;CURRENT LOCATION
0028           02130 BUFF     DEFS     40             ;WORD AREA
91C8 00        02140          DB       0
9000           02150          END      ADVENT
00000 Total errors
```

The ADVENT routine is the execution control loop, CALLing the various sections of the program in correct sequence. Interrupts are disabled (to make sure no other routine cuts into execution) and the stack pointer is moved to the top of 64K RAM. The origin of the program and the location of the stack pointer will depend on the available memory in your system. I recommend that you have access to at least 10K of RAM or else break the program up into small segments if you intend writing programs of this nature.

The DECODE routine works like the SEARCH routine. The routine starting at NEXT1 is basically a repeat of DECODE except this time designed to identify the second word. The program could have been written to CALL this type of routine twice, but has been divided into two sections for clarity. The routines SCROLL and GKEY must be modified to suit the appropriate computer. As is, the program accepts one verb – LOOK and one noun – WINDOW (shortened in the program to WINDO), and is ready to be expanded with further data.

This section is kept separate from chapter five, while still called a 'game', to emphasize the value of this type of programming. It is a worthwhile exercise in string manipulations, arrays, jump tables and data organisation, and is therefore – unlike programs in the next chapter – not written solely for the sake of 'fun'. This type of word decoding generates *tokens,* numbers that represent more complex symbols to save space and make manipulations easier. Such processing is an essential part of any computer language.

Firstly, the listing does not include a GET/DROP routine that will keep track of all items, or code allowing movement. In an adventure the operator is allowed to pick up, drop and manipulate items (feel, touch, break, etc.), and move from one location to another. To do this a few other routines are required. One that will 'pick' items, 'drop' them again, take an inventory of them, allow the operator to 'move around', and finally, to display entrances and exits. The rest is mainly data, specifying what can be touched, what clues are to be given, what is needed to get certain items, and possible locations.

The GET/DROP/INVENTORY routines must access a table to locate information about each item. Turning your attention to the listing again, notice that once the first word is matched, a jump is made to a specific routine. The second word, the noun, if it can be matched, is assigned a number according to its location in TABLE2.

A special byte must be assigned to each item used in the game to give it a location. By comparing the ROOM value with the value in the table, you can calculate if the item is in the current location. It's unlikely that you will have more than one or two hundred locations at the most, so byte 255 is used as a 'I'm carrying it' indicator. The following is an INVENTORY routine:

```
INVEN   LD      HL,M1           ;'FOLLOWING:'
        CALL    PRINT           ;DISPLAY,
        LD      B,10            ;10 ITEMS
        LD      IX,GTABLE       ;ITEMS LOCATION
        LD      HL,NTABLE       ;ITEMS NAME
LOOP    LD      A,(IX)          ;GET LOCATION
        INC     A               ;IS IT 255?
        PUSH    AF              ;SAVE FLAGS
        CALL    Z,PRINT1
        POP     AF              ;RECOVER FLAGS
        CALL    NZ,SKIP0        ;NEXT NAME
```

```
        INC     IX              ;NEXT ITEM
        DJNZ    LOOP
        JP      ALP             ;FINISHED
SKIP0   LD      A,(HL)          ;SKIP NAME
        INC     HL              ;NEXT BYTE
        CP      '#'
        RET     Z               ;NEW NAME?
        JR      SKIP0           ;NOT YET
GTABLE  DB      4,21,11,1,2,4,1,19,5,5
NTABLE  DB      'OLD BOOK#IRON ROD#'
        DB      'TOOTHPICK#BROKEN GLASS#'
        DB      'MIRROR#BREAD KNIFE#SMALL
        DB      ' BLANKET#BOX OF MATCHES#'
        DB      'BROKEN MIRROR#'
```

Which displays all items being carried. While each noun has an abbreviated form: BOOK/ROD/TOOT/KNIF, it also has a 'complete' name. The PRINT1 routine differs slightly from PRINT.

```
PRINT1  LD      DE,PNT
PRINT2  LD      A,(HL)
        CP      '#'
        JR      Z,SKIP1
        LD      (DE),A
```

```
SKIP1   INC     HL
        INC     DE
        CP      '#'
        JP      Z,SCROLL
        JR      PRINT2
```

To GET an item it must be in the current location and equal to the number in ROOM. Then a 255 ('being carried' code) replaces the byte in GTABLE.

```
GET     LD      IX,GTABLE-1
        CP      10              ;ONLY TEN ITEMS
        JR      NC,IMPOS        ;"CAN'T MOVE"
        LD      E,A             ;POINT TO ITEM
        LD      D,0
        ADD     IX,DE
        LD      A,(IX)          ;GET IT
        INC     A               ;IS IT 255?
        JR      Z,IMPOS1        ;"I HAVE IT"
        LD      C,A             ;COMPARE ROOM
        LD      A,(ROOM)
        CP      C
        JR      NZ,IMPOS2       ;'NOT HERE'
        LD      (IX),255        ;GOT IT
        LD      HL,OK           ;'OK'
```

```
        CALL    PRINT
        JP      ALP
OK      DB      'OK',Ø
```

The labels IMPOS, IMPOS1, and IMPOS2 display the messages 'I CAN'T GET THAT!', 'I ALREADY HAVE IT' and 'I DON'T SEE IT HERE' or something to that effect. Some items can be manipulated but not moved, like a door or bookshelf. Items that can be picked up go into the table first. Anything past a specified number, ten in this example, will RETurn with a message telling you that it cannot be moved. The DROP routine is the reverse of GET:

```
DROP    LD      IX,GTABLE-1
        CP      1Ø              ;NOT POSSIBLE
        JR      NC,IMPOS
        LD      E,A
        LD      D,Ø
        ADD     IX,DE
        LD      A,(IX)
        INC     A               ;CP 255?
        JR      NZ,IMPOS2
        LD      A,(ROOM)        ;GET LOCATION
        LD      (IX),A          ;PUT IN TABLE
        LD      HL,OK
        CALL    PRINT
        JP      ALP
```

The routine checks, by looking for 255, that you GET the item and then changes its value in the GTABLE to that of the current location.

Each location in this adventure is assigned a byte. Of this, bits 0-3 are used to indicate the four valid directions; north, south, east and west. Since the left nibble is not used, it can store other special information about each location. Conceivably an 'up' and 'down' or other types of entrances and exits could be added (or bit 6 could be set to indicate that the room is radioactive for that matter). To display this information:

```
DIR     LD      A,(ROOM)
        CALL    GROOM           ;GETS ROOM BYTE
        BIT     0,A
        CALL    Z,PNO           ;DISPLAY NORTH
        BIT     1,A
        CALL    Z,PSO           ;DISPLAY SOUTH
        BIT     2,A
        CALL    Z,PEA           ;DISPLAY EAST
        BIT     3,A
        CALL    Z,PWE           ;DISPLAY WEST
        JP      SCROLL
```

The routines PNO, PSO, PEA and PWE display the north, south, east, and west messages on the VDU. Naturally dressing up like a 'POSSIBLE EXITS ARE:' message should be added in the completed game. Note that the bit in the location byte must be reset to make a particular direction a valid one. Finally the routine exits by scrolling the VDU.

Possible entrances and exits are stored in memory as a two dimensional array. To access this array call GROOM:

```
GROOM   LD      HL,MTABLE       ;ARRAY
        LD      B,A
```

```
        PUSH    DE
        LD      E,A
        LD      D,Ø
        ADD     HL,DE
        LD      A,(HL)
        POP     DE
        RET
```

To move north, ten bytes are subtracted from the current *y* axis of the array. To move south, ten bytes are added. Movement east increments the *x* axis of the array, westward movement decrements it. The move is invalid if the bits in the current location disallow it. So, it is possible to create walls and other barriers, exits and entrances in map fashion, with each location a square on grid paper, by setting or resetting bits 0-3 in each byte.

```
NORTH   LD      A,(ROOM)
        SUB     1Ø              ;MOVE NORTH
        CALL    GROOM
        BIT     Ø,A             ;MOVE VALID?
        JR      NZ,ERROR
EXIT1   LD      A,B
        LD      (ROOM),A        ;SAVE NEW LOCAT
        LD      HL,OK
        CALL    PRINT
        JP      ALPØ
```

```
SOUTH   LD      A,(ROOM)
        ADD     A,10            ;MOVE SOUTH
        CALL    SETUP
        BIT     1,A             ;MOVE VALID?
        JR      NZ,ERROR
        JR      EXIT1
EAST    LD      A,(ROOM)
        INC     A               ;MOVE EAST
        CALL    SETUP
        BIT     2,A             ;MOVE VALID?
        JR      NZ,ERROR
        JR      EXIT1
WEST    LD      A,(ROOM)
        DEC     A               ;MOVE WEST
        CALL    SETUP
        BIT     3,A             ;MOVE VALID?
        JR      Z,EXIT1
ERROR   LD      HL,NOMVE        ;'NO EXIT'
        CALL    PRINT
        JP      ALP

NOMVE   DB      'THAT IS NOT AN EXIT!',0
```

CHAPTER 5
– THE COMPUTER GAME

Game writing is one of the few areas of programming where coding skills and imagination are put directly and furiously to the test. Beyond the essential mechanics of machine language there are very few rules for 'game programming'. This chapter will look at processes and techniques that make programming a little bit more bearable for your writer.

The algorithms behind animation, random number generating, maze generating, and arcade games are of course essential. But they require an understanding of patterns, coordinate geometry, and randomness. Further, it is necessary to sort and organise information, simulate real-world effects (a bouncing ball), appreciate the importance of sequence and data, and simultaneous processing. A book itself could be dedicated to games, but that must wait for the future. This chapter deals with the basic principles of *commercial* game programming.

How big is a game? How long does it take to write one? A game can take a few hours to write and be anything less than a thousand bytes to something that spans hundreds of Ks and years of labour. Invader type games have been written in under one K, others, in over a hundred. On a personal computer quality games can be written in three or four thousand bytes, with few exceeding 10K as a reasonable average. Fifty or sixty hours of work, with experience, is an acceptable period of time for a good piece of coding.

5.1 RANDOM NUMBER GENERATING

Games have to be partially indeterminate, or unpredictable, to be truely enjoyable and relieve creeping monotony. Games have been written, and can still be written, that require no random processing at all. The computers move can depend solely on the move of the player, needing no randomness at all. A game written on this premise can be very complex, but must nevertheless repeat in some sort of regular way.

Can a computer generate a genuine random number? The answer is 'no', but it can seem like it, which is good enough. A typical 'random number generator' is a series of additions, multiplications, divisions, and shifts, etc. The remainder of the operation is then made the subject of the next calculation. The following equation illustrates this:

$$R = (B \times R + C) / D$$

Where B, C and D are predefined constants. In machine language this equation could take the form of:

```
RND     ADD     A,A             ;R= ACCUMULATOR
        DJNZ    RND             ;B= B REGISTER
        ADD     A,C             ;C= C REGISTER
RND1    SUB     D               ;D= D REGISTER
        JR      NC,RND1         ;NO OVERFLOW
        ADD     A,D             ;GET REMAINDER
```

On exit the accumulator has been multiplied by B, added with C and divided into (by multiple subtracts) D. The result can be put back into the equation to yield *another* result. For example, if we loaded B with 5, C with 7 and D with 9 and CALLed RND six times we would generate the numbers:

7 6 1 3 4 0

in apparent random sequence. These numbers will repeat indefinitely if the routine is CALLed further. If we used larger values, it would yield a larger succession of numbers. Loading B with 59, C with 71 and D with 93 would generate:

71 75 32 6 53 36 56 27 83 39

47 54 2 3 62 9 44 63 68 84

5 87 89 21 8 78 23 33 65 0

before the pattern repeats (but we would have to do the maths using register pairs to keep the values in calculable range).

These are not really random numbers at all. Just a sequence of predefined results seeming at first glance to be random. Some register values yield very poor number patterns. Some will only display a series of zero or seven or something similar. Note: With both examples odd numbers were used, since a 'remainder' is needed as a seed for the next operation.

Large numbers, up to 65535 and even over, can be used by changing the routine to handle 16-bit, 32-bit or 64-bit numbers.

The longer the string of numbers of course, the longer the seeming randomness. The process nevertheless inevitably repeats itself. Invader type games find this method ideal, since the 'aliens' are expected to follow some set pattern anyway (there are few games without a pattern, intentional or not). Two random number generators could be used together, the first generating the constants of the second. The random sequence could, seemingly, go on forever.

The remaining problem is one of order. The random numbers will always be the same random numbers (not really very random) when the game restarts. Which might be a bit too predictable for computer poker or blackjack . . . (The 'seed' for a random number generator could be derived from user input. A counter being constantly incremented is stored away as the new 'seed' at the moment the user presses the 'continue' key. This is not really computer randomness however, but operator randomness!). The architecture of the Z80 offers a solution. Examine the following instruction:

```
        LD      A,R
```

which loads the accumulator with the refresh counter, used for the internal maintenance of RAM chips. An automatic procedure the programmer normally does not worry about. The contents of R, since it's a counter, is changing constantly. Jumping between zero and 127 (7-bits) at any given moment. Actually this looping of the refresh register is only pseudo-random. It too, like the equation already shown, follows a preset sequence. This however only occurs if *and only if* the instructions executed in the program are identical. For example:

```
        LD      B,100
RND     LD      A,R
        LD      (IX),A
        INC     IX
        DJNZ    RND
```

will readily produce a pattern. In many games no definite set of instructions are executed every time the major subroutines are CALLed. Some instructions are only executed once every two CALLs or only occasionally. The larger the program the more R loses any regular pattern. In other words, R should be used, albeit sparingly, well spaced throughout the program.

If careful attention is payed to these limitations, the use of R as a random result is ideal for most purposes. In generating random mazes, R could be made solely responsible for billions of different patterns.

The following subroutine was used successfully in a popular arcade game to handle all its random processing:

```
;8-BIT RANDOM NUMBER GENERATOR
;ENTRY -->    C  = 20-127
;EXIT  -->    A  = RANDOM NUMBER
;                ALL OTHER REGISTERS SAVED
RNDA    LD      A,R             ;REFRESH RANDOM
        CP      C               ;RANGE
        RET     C               ;RET IF SMALLER
        SRA     A               ;TRY AGAIN
        CP      C               ;ANY GOOD?
        RET     C
        PUSH    BC              ;SAVE BC
        LD      B,A             ;DELAY FOR R
RNDLP   DJNZ    RNDLP
        POP     BC
        JR      RNDA
```

Numbers less than the recommended parameters can be put in C, but there is the risk of 'locking-up' the computer while the routine waits for a mistimed R; but such a thing happening is unlikely. The program RETurns a result in the 'range', but not exceeding, C. If C is loaded with 50, only values less than 50 are RETurned. This feature is a useful one. After checking for range, the computer shifts the bits into a new position, and checks the range again. If it still exceeds the

value of C, it executes a delay which partially compensates for the limitations of accessing the refresh register. The process repeats until a valid number is returned.

An obvious example of randomness is dice rolling. Using R as the only source of randomness, we get a reasonably (some numbers will appear more than others) random result. To execute, CALL RDICE:

```
NOTIT   DJNZ   NOTIT
RDICE   LD     A,R
        LD     B,A
        AND    7
        CP     7
        JR     Z,NOTIT
        OR     A
        JR     Z,NOTIT
        RET
```

which returns a number between one and six. For yes/no decisions it is easier to:

```
        LD     A,R
        CP     64
        JR     C,YES
```

rather than go to the trouble of writing a special routine.

The final routine is a true pseudo-random number generator. The 'seed' for the random result must be periodically generated elsewhere in the program by using the R register. This will make the generator so random that 'pseudo' seems slightly unfair. Examine the listing:

```
;PSEUDO-RANDOM 16-BIT NUMBER GENERATOR
;ENTRY -->    NO CONDITIONS
;EXIT  -->    RANDOM NUMBER IN BC

RNDBC  LD     DE,(SEED)
       LD     HL,(SEED1)
       LD     B,5          ;SHIFT 5 TIMES
LOOP   RR     H
       RL     L
       RR     D
       RL     E
       DJNZ   LOOP
       LD     B,3          ;SUB 3 TIMES
LOOP2  PUSH   DE
       LD     DE,(SEED1)
       OR     A
       SBC    HL,DE
       EX     DE,HL
       POP    HL
```

```
        DJNZ    LOOP2
        LD      (SEED),DE   ;SAVE
        LD      (SEED1),HL  ;SAVE
        LD      B,E         ;GET NUMBER
        LD      C,H
        RET

SEED    DW      2812        ;TYPICAL SEEDS
SEED1   DW      3488
```

The routine rotates the registers D, E, H and L five times, and then performs multiple subtracts. The routine is fairly flexible so try experimenting with it.

Variations on the principles discussed will produce just about any random number generator to suit your needs.

5.2 MAZE GAMES

Anyone familiar with computer games should be familiar with the notorious maze of doom. Whether this is the fabled labyrinth beneath Knozzos – complete with fabled, and ravenous, minotaur, or merely little creatures of unknown origin, after your 'gobbler', maze generating is one of the foundations of game programming. Video games use mazes as 'maps' to guide the player through a moonbase filled with robots, or a castle of ghosts. And, of course, strategy games like *Dungeons and Dragons* need them to create new scenarios. Whether you intend writing a maze game or not, you will probably find the need for one. This section looks at the maze generating algorithm in detail, and the principles of three dimensional maze drawing.

A true maze has only one entrance and one exit, with only one more or less direct – route connecting both. A corridor must occupy every available portion of the maze. They can intersect vertically or horizontally, but cannot loop into one another. In other words, there is a valid way to get to every part of the maze.

Examine diagram 5.0, a typical maze. Moving the entrance or exit up or down the 'walls' two units at a time (a 'unit' is a corridor width, called a 'cell'), will create a new direct route from entrance to exit. The exit and entrance can go anywhere, in steps of two cells, but are usually put on opposite 'walls' to make the maze more difficult to traverse.

Observe that the white and black cells keep their distance, unless horizontally or vertically joined together, such that the white cells are always divided by the black cells.

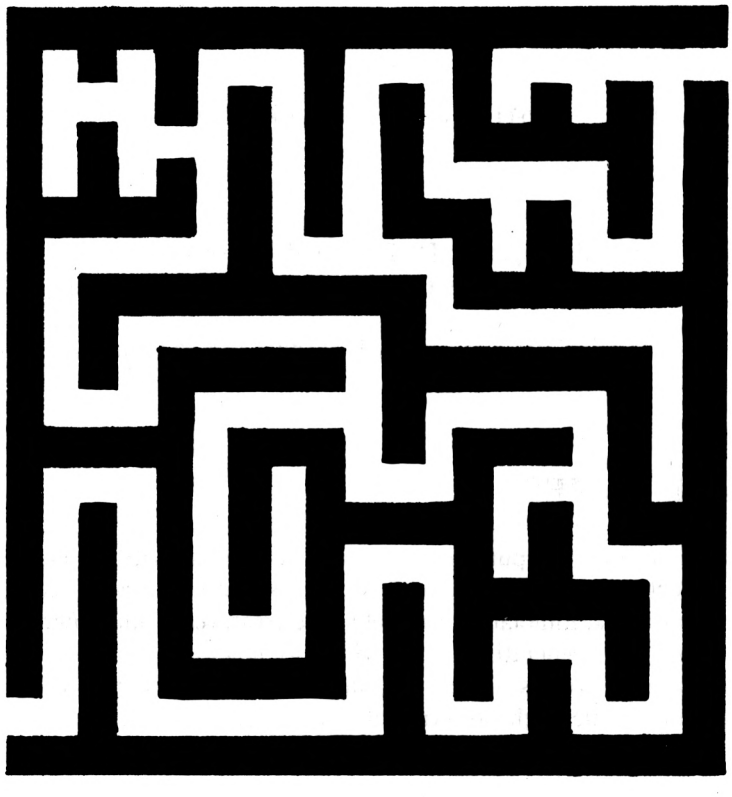


DIAGRAM 5.0 – TYPICAL MAZE STRUCTURE.

The algorithm for generating a maze is not complex, but difficult to visualize. You may have to read this section over a few times and scrutinize the routine before you grasp the principles involved.

Basically, a maze generating routine is made up of two parts. The first must check all four directions; north, south, east, and west, and determine if a passage has already been drawn there. In diagram 5.0 the walls are black, the corridors white. Anything black is therefore in an 'unvisited' state. The routine must exit with a byte indicating the state of the four potential exits. A zero byte indicates no valid direction.

The second part of the program must select randomly from its four, or less, possible choices, and then draw a piece of corridor leading from the current location to the new location.

When searching for an 'unvisited' cell (anything black), the program sets up a 'pointer' that marks the part of the maze the program is creating. This is then inverted (changed from black to white). It then scans not one cell ahead, but two. If this cell is 'unvisited', the program inverts it as well. Provided that we are using one byte to represent one cell, we would have two white 'spots' one byte apart. The program must therefore invert the middle cell – the cell between the current pointer and the new cell. A little corridor, of three cells, is consequently created. The program takes the cell it had scanned as its new pointer and the scan continues all over again.

Now, what happens if the randomness of the selection means that there are no more 'black' cells around the new pointer? Take a piece of paper and a pen, and start drawing a line. In this case the corridors, your pen, is black, and the 'unvisited' cell, the paper, white. Move in random directions but do not allow yourself to cut over any other line or come within one centimetre of a line (to simulate width).

The result? Very soon you cut yourself off from other sections of the paper. There are still white spots on the paper, but you cannot reach them because this requires you to cut across another line or come unreasonably close. The problem is solved by taking your pen off the current line, finding a previous line, and branching out from this one. You will end up creating a random maze.

A random maze will cut itself off many times in the drawing process. What it does is move its 'pen' left or right or up or down to a cell that has *already* been 'visited'. It then looks for any 'unvisited' white cells. It continues to move to new cells until at least every 'visited' cell has been checked to see if it borders an 'unvisited' one. The majority of the maze is drawn up quickly. The rest of the time taken is used looking for all the leftover, 'unvisited' cells.

Listing 5.0 is a complete maze generating program.

```
                00100 ;2D MAZE GAME GENERATOR
                00110 ;USE 1 BYTE FOR EACH CELL
                00120 ;*MUST CHANGE ACCORDING TO COMPUTER
                00130 ;**********************************
                00140 ;
3C00            00150 VDU       EQU     15360           ;*START OF VDU RAM
03FF            00160 LEN       EQU     1023            ;*LENGTH OF VDU RAM
00BF            00170 WHITE     EQU     191             ;*ALL WHITE GRAPHIC BLOCK
0080            00180 BLACK     EQU     128             ;*ALL BLACK GRAPHIC BLOCK
0040            00190 COLUMN    EQU     64              ;*NO. OF COLUMNS
9000            00200           ORG     9000H           ;*
9000 F3         00210 MAZE      DI                      ;SETUP RAM & REGISTERS
9001 31FFFF     00220           LD      SP,65535        ;*TOP OF 64K RAM
9004 21003C     00230           LD      HL,VDU          ;CLEAR VDU
9007 11013C     00240           LD      DE,VDU+1
900A 01FF03     00250           LD      BC,LEN
900D 36BF       00260           LD      (HL),WHITE
900F EDB0       00270           LDIR
9011 01B400     00280           LD      BC,180          ;*SIZE = HEIGHT x WIDTH
9014 ED430391   00290           LD      (COUNT),BC
9018 210820     00300           LD      HL,2008H        ;*HL= X,Y POINTERS
901B CDF390     00310           CALL    ZAP             ;DRAW STARTING PLACE
901E CD2F90     00320           CALL    RUN             ;DRAW MAZE
9021 210C01     00330           LD      HL,010CH        ;*DRAW ENTRANCE
9024 CDF390     00340           CALL    ZAP
9027 21023D     00350           LD      HL,3D02H        ;*AND EXIT
902A CDF390     00360           CALL    ZAP
902D 18FE       00370           JR      $               ;FINISHED!
902F ED4B0391   00380 RUN       LD      BC,(COUNT)      ;EXECUTION CONTROL LOOP
9033 0B         00390           DEC     BC              ;DECREMENT NO. OF CELLS
9034 ED430391   00400           LD      (COUNT),BC
9038 78         00410           LD      A,B
9039 B1         00420           OR      C
903A C8         00430           RET     Z               ;DONE?
903B CD4690     00440           CALL    FIND            ;FIND POSSIBLE EXITS
903E CDA590     00450           CALL    RND             ;SELECT FROM CHOICE
9041 CDBE90     00460           CALL    BUILD           ;DRAW IT
9044 18E9       00470           JR      RUN             ;AND AGAIN
9046 DD21FB90   00480 FIND      LD      IX,TABLE        ;INSTRUCTION TABLE
904A FD21FF90   00490           LD      IY,EDGE         ;BORDER OF MAZE
904E 010004     00500           LD      BC,400H         ;B=64, C=0
9051 CB21       00510 FLP       SLA     C               ;MOVE TO NEXT BIT
9053 E5         00520           PUSH    HL              ;FIND DIRECTION TO,
9054 DD7E00     00530           LD      A,(IX)          ;CHECK
9057 325D90     00540           LD      (INST),A
905A 325E90     00550           LD      (INST+1),A
905D 0000       00560 INST      DW      0               ;CHECK IT
905F 1F         00570           RRA                     ;CHANGE INC/DEC,
9060 1F         00580           RRA                     ;TO "LD" INSTRUCTION
9061 1F         00590           RRA
9062 F678       00600           OR      78H             ;01111000B
9064 326790     00610           LD      (INST2),A
9067 00         00620 INST2     NOP                     ;LOAD IT
9068 FDBE00     00630           CP      (IY)            ;OVER EDGE?
906B 2807       00640           JR      Z,PASS          ;YES?
906D CDED90     00650           CALL    CONV            ;CALC X,Y
9070 2002       00660           JR      NZ,PASS         ;NO MOVES?
9072 CBC1       00670           SET     0,C             ;YES
9074 E1         00680 PASS      POP     HL              ;RECOVER HL
9075 DD23       00690           INC     IX              ;ADJUST POINTERS
9077 FD23       00700           INC     IY
9079 10D6       00710           DJNZ    FLP             ;AGAIN
907B AF         00720           XOR     A               ;IS C ZERO?
907C B9         00730           CP      C               ;NO MOVES POSSIBLE?
907D C0         00740           RET     NZ
907E FD21FF90   00750           LD      IY,EDGE
```

```
9082 24        00760 MOVE     INC     H               ;TRY ANOTHER PART,
9083 24        00770          INC     H               ;OF MAZE
9084 7C        00780          LD      A,H
9085 FDBE00    00790          CP      (IY)            ;OVER EDGE?
9088 2807      00800          JR      Z,FIX           ;ADJUST IF OVER
908A CDED90    00810 MOVE2    CALL    CONV            ;HAS IT,
908D 20B7      00820          JR      NZ,FIND         ;BEEN VISITED?
908F 18F1      00830          JR      MOVE            ;NO, KEEP SEARCHING
9091 FD6602    00840 FIX      LD      H,(IY+2)        ;MIN X VALUE
9094 24        00850          INC     H               ;LOOK ELSEWHERE
9095 24        00860          INC     H
9096 2C        00870          INC     L
9097 2C        00880          INC     L
9098 7D        00890          LD      A,L             ;WATCH FOR MAZE LIMITS
9099 FDBE01    00900          CP      (IY+1)          ;OVER EDGE?
909C 20EC      00910          JR      NZ,MOVE2
909E FD6E03    00920          LD      L,(IY+3)        ;THEN GIVE IT MIN Y VALUE
90A1 2C        00930          INC     L
90A2 2C        00940          INC     L
90A3 18E5      00950          JR      MOVE2           ;TRY NOW
               00960 ;
90A5 ED5F      00970 RND      LD      A,R             ;RANDOM NO. ROUTINE
90A7 E60F      00980          AND     15              ;00001111B
90A9 47        00990          LD      B,A
90AA 3E02      01000          LD      A,2
90AC CB2F      01010 RSHIFT   SRA     A               ;SPIN "A" AROUND
90AE 3002      01020          JR      NC,RSKP
90B0 CBDF      01030          SET     3,A
90B2 10F8      01040 RSKP     DJNZ    RSHIFT
90B4 47        01050 BACK     LD      B,A             ;SAVE A IN B
90B5 A1        01060          AND     C               ;RESULT NO GOOD?
90B6 2802      01070          JR      Z,RND1
90B8 4F        01080          LD      C,A             ;EXIT, C=BIT 0,1,2,3 SET
90B9 C9        01090          RET
90BA 78        01100 RND1     LD      A,B             ;RECOVER A
90BB 0F        01110          RRCA                    ;FIX RESULT
90BC 18F6      01120          JR      BACK            ;TRY NOW
               01130 ;
90BE DD21FE90  01140 BUILD    LD      IX,TABLE+3      ;MUST BE BACKWARDS NOW
90C2 DD7E00    01150 BLP      LD      A,(IX)          ;GET INSTRUCTION
90C5 32D390    01160          LD      (SET),A
90C8 32D790    01170          LD      (SET1),A
90CB CB19      01180          RR      C               ;CHECK BIT
90CD 3804      01190          JR      C,SET           ;BIT SET?
90CF DD2B      01200          DEC     IX              ;NO
90D1 18EF      01210          JR      BLP             ;KEEP TRYING
90D3 00        01220 SET      NOP                     ;INC/DEC GOES HERE,
90D4 CDF390    01230          CALL    ZAP
90D7 00        01240 SET1     NOP                     ;AND HERE
90DB 1819      01250          JR      ZAP             ;DRAW IT, AND RETURN
               01260 ;
               01270 ;SUBROUTINE PACKAGE:
90DA E5        01280 CALC     PUSH    HL              ;CALCULATE X,Y OF HL
90DB D9        01290          EXX
90DC E1        01300          POP     HL
90DD 45        01310          LD      B,L
90DE 4C        01320          LD      C,H
90DF 114000    01330          LD      DE,COLUMN
90E2 21C03B    01340          LD      HL,VDU-COLUMN
90E5 04        01350          INC     B               ;MUST NOT BE ZERO
90E6 19        01360 CLP      ADD     HL,DE
90E7 10FD      01370          DJNZ    CLP
90E9 09        01380          ADD     HL,BC           ;CALCULATE X
90EA 7E        01390          LD      A,(HL)          ;GET BYTE AT X,Y
90EB D9        01400          EXX
90EC C9        01410          RET
90ED CDDA90    01420 CONV     CALL    CALC            ;GET X,Y OF HL
90F0 FEBF      01430          CP      WHITE           ;IS IT "ON"?
90F2 C9        01440          RET
90F3 CDDA90    01450 ZAP      CALL    CALC            ;GET X,Y OF HL
```

```
90F6 D9          01460          EXX
90F7 3680        01470          LD      (HL),BLACK      ;DRAW "PASSAGE"
90F9 D9          01480          EXX
90FA C9          01490          RET
90FB 24          01500 TABLE    DB      24H,2CH,25H,2DH ;INC H, INC L, DEC H, DEC L
     2C 25 2D
90FF 3E          01510 EDGE     DB      62,14,0,0       ;*XMAX,YMAX,XMIN,YMIN EDGE
     0E 00 00
9103 0000        01520 COUNT    DW      0               ;COUNTER VARIABLE
9000             01530          END     MAZE
00000 Total errors
```

The TRS-80's simple design comes in handy to demonstrate this program. The stack is pushed to the top of 64K RAM and the 1K of VDU RAM (addresses 15360-16383) is 'whited' out (our piece of paper) with an all-white graphic block; byte 191 using this computer.

The register pair BC is loaded with 180, the number of cells to be drawn. The program treats VDU RAM as a two dimensional matrix, 64 columns by 16 rows – exactly 1K. Exactly half of RAM will be occupied by corridors and exactly half by walls. This leaves 32 columns by 8 rows of cells to be 'visited'. We want to keep the edges of the maze white, so by moving the maze in two bytes; 30 columns by 6 rows, we are left with the calculable number of cells to 'visit'. That is: height * width; 30 * 6 or 180 cells.

The register pair HL acts as the ARRAY(x,y) pointers. Register H acts as x and L as y. It is loaded with the starting position ARRAY(32,8) which is roughly the middle of VDU RAM. It is irrelevant where the pointers are placed (as long as it is inside the array of course) and any location will do. The program then CALLs 'RUN' which draws the maze. After drawing the maze, HL is loaded with the bottom left hand corner of the maze and an exit is created. It does the same thing again, except placing the exit in the top right hand corner, making the two openings as physically far apart as possible. The entrance and exit can be placed anywhere, provided the x value of ARRAY(x,y) is even. Had the original pointers been given odd numbers, the exits would also need odd x co-ordinates.

The routine 'RUN' executes the two important routines discussed; FIND possible directions, and BUILD a passage. The counter, keeping track of cells 'visited', is decremented and FIND is CALLed first.

In the FIND routine, IX is loaded with the instruction table. It holds four bytes with the hexadecimal codes 24, 2C, 25 and 2D, equivalent to the Z80 instruction code INC H, INC L, DEC H and DEC L. Register IY points to the EDGE table. This contains the ARRAY(x,y) maximum and minimum values, and is used to check that adjustments of the pointer do not exceed the maze's border. Register B is loaded with four, the number of directions to check, and C is zeroed. The routine gets the instruction out of IX, and places it in INST and the address after. INST is executed and the pointer is adjusted two bytes in one of the possible directions.

(Remember: the maze program must always 'look' two cells ahead, if suitable, clear the new cell and the cell in between the two others.)

The instruction loaded in A is converted from an 'INC' to a 'LD A,' by shifting the bits in A right three times, and ORing 78H. Why does this happen? A full explanation is too complex for this chapter, but it will be dealt with further in chapter seven. For now accept that an INC H is converted into a LD A,H or a DEC L is converted into a LD A,L. The instruction is then executed.

The new pointers in HL are checked to make sure they do not exceed the edges of the maze. If they do not, CONV is CALLed. It returns with the Z flag set if the cell has not been 'visited'. If Z is set, a bit is set in C. The process repeats four times, checking all four directions, and exiting with a number between 15 (00001111B) and zero. If non-zero the RET is executed.

If no move is possible, either because all the cells have already been 'visited', or it would mean going beyond an edge of the maze, MOVE is executed. This routine moves the HL pointers around VDU RAM looking for a previously *'visited'* cell that borders an 'unvisited' one. Eventually it will find one.

The routine RND, short for RaNDom, selects a direction from the four or less bits set in C. The accumulator is loaded with one bit set, and rotated around. It is then ANDed with C. If it cancels out, the move is invalid and the value in A is recovered and shifted again. Eventually one direction is chosen.

The routine BUILD loads IX with the instruction table, this time backwards to correspond with the bit position in C. The register is rotated right, with bit 0 moved into the carry flag, until it becomes set. By this time lines 990-1010 have set up the INC or DEC instructions in SET and SET1. The HL pointer is adjusted and the routine ZAP inverts a cell; used in drawing the corridor (refer to the next section regarding VDU RAM loading).

The remaining routine CALC is used by ZAP and CONV to convert the matrix back into a linear array for reading or loading bytes.

Bytes are horrible things when put on VDU RAM. It is more than likely that a completely 'white' or completely 'black' graphic block looks narrow or flat rather than perfectly square when it appears on your VDU, so this irregularity will be reflected in the overall appearance of the maze. This problem can be fixed by grouping sets of bytes, or even bits, together to represent square blocks. To do this you must change the ZAP and CALC routines to suit your machine. Of course, the maze program is only an example, and has been made very unspecific.

The program currently generates mazes on its most difficult level. Simpler mazes can be generated by CALLing the RND routine less often. For example, a simple level maze would CALL RND once every five corridors had been drawn, or when it must select a new direction to draw in. This way, corridors wrap and turn less, and are easier to follow.

The only thing better than a 2D maze is a 3D maze.

This step is not as difficult as it may sound. Height and perspective are only illusions, created by the effective use of graphics.

The procedure is as follows: Create a conventional 2D maze somewhere in RAM. Now create a picture of a corridor receding into infinity – use your imagination here. To help you, examine diagram 5.1. There should be a pointer keeping track of a location in the 2D maze as well as a compass-point direction you (your alter-ego in the maze) are facing.

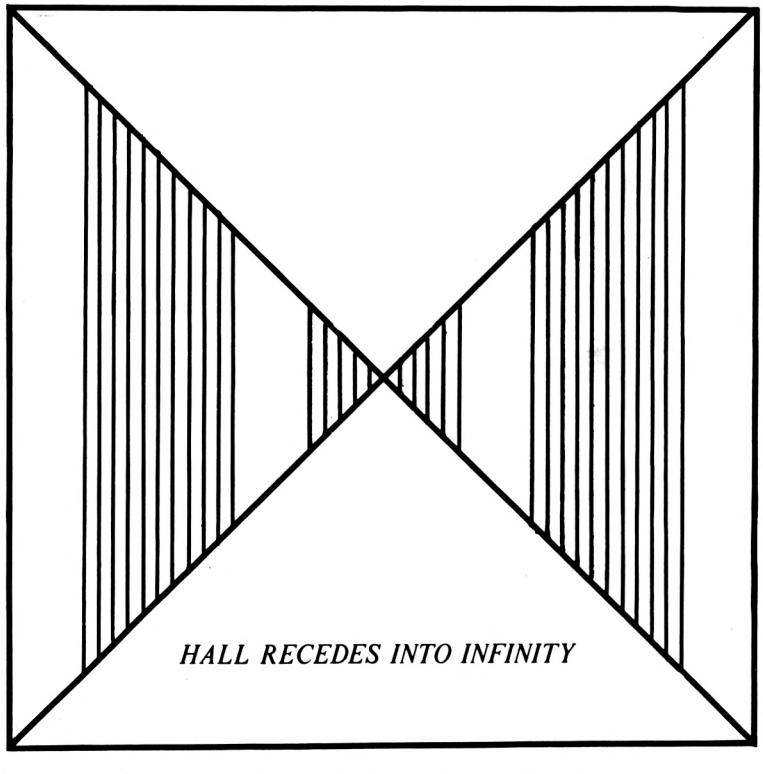


DIAGRAM 5.1 – CONSTRUCTING THREE DIMENSIONAL MAZES

Your receding corridor is a visual image of your 2D maze, say, four cells ahead. The first part of your routine must check if the passage, according to the direction you are facing, continues for four cells or more. It it doesn't, you have to draw a 'dead end' over the front part of the receding corridor. See diagram 5.2. The dead end's size and location must be measured carefully, and be in proportion relative to the actual distance of the corridor's end.

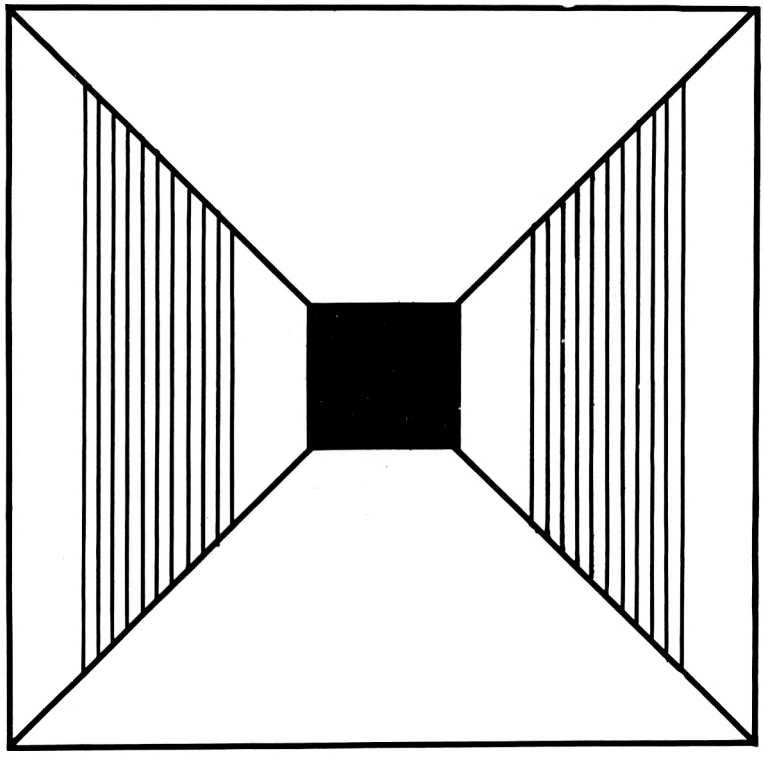
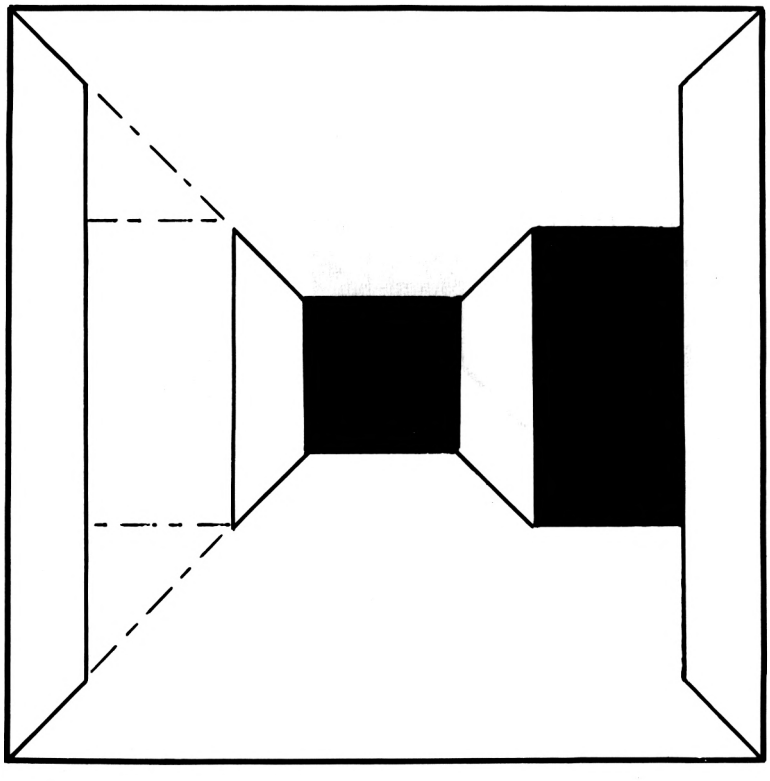


DIAGRAM 5.2 – OVERLAPPING GRAPHIC BLOCKS CREATE DEAD END.

You must also check left and right of each cell for an exit. If it finds one, the program must 'cutaway' a section of corridor (again, according to scale). See diagram 5.3. That's all there is to it. A routine to scan forward, left and right, and return bits set or reset, and another routine to display the 'dead ends' and cut away corridors that intersect with the main passage according to these bits.

With one of my programs based on this principle, I used only two-thirds of VDU RAM for the receding corridor. The rest I filled with other information; including an indicator that showed the direction I faced, and a little 'scanner' that gave a 2D display of a portion of the maze with me in the centre. A blinking dot constantly marking my position. The maze can then be filled with killer robots or ghosts or racing cars or mice or whatever you like, or, just make it a simple 'escape alive' game. The 3D maze game can be combined with an adventure with very interesting results.



5.3 COORDINATE GEOMETRY

Visual programs like mazes or arcade games depend on graphics, and graphics depend on VDU RAM. While your Z80 treats VDU RAM just like any other piece of memory, it's fairly obvious that the rest of your computer sees it very differently. If your system runs CP/M then you probably have the '80 column' standard. If you have a small colour computer, you probably have 32 or 40 columns to the line.

We are all familiar with 'wrap-around' and 'scrolling' which help to demonstrate two principles. A VDU 'scrolls' because a program has block moved RAM up or down the column length. On the other hand, 'wrap-around' is a built-in feature of the hardware. You cannot stop it from happening. This presents a problem to the programmer; what appears to be a two dimensional array on the VDU is merely a linear array. The solution is to treat it as a two dimensional array, enabling us to work with two dimensional coordinates. This technique has already been used in the previous section, but we will now deal with it – and VDU RAM, in greater depth.

Before we continue it should be mentioned that many modern computers have a very sophisticated VDU RAM configuration. On the majority of computers, the following instruction is valid:

```
LD      (VDU),'A'
```

which will put the letter 'A' in (the label VDU represents the first location of VDU RAM) the top left hand corner of the VDU. However, an ever increasing number of computers are being produced with what is termed 'arcade quality' graphics. This is wonderful for the arcade game player, but can fill a programmer with dread and loathing. The VDU is, in this case, not fully memory mapped because the pixel (PIcture ELement) resolution is too high. Memory addressable RAM is too expensive, except for quality business systems, or systems dedicated to game play only, so other techniques have to be used. Some systems reserve 16K of RAM just for the VDU, and then require separate data tables and OUT instructions to access it.

There is no answer to this problem. The programmer must refer to the available technical information and write routines that will allow access to this RAM in some comprehensible manner. Such programming is far beyond the scope of this book, so rather than tackling the problem, I will pass over it with only a few comments. To compensate for this, I will, for the most part, concentrate on the mechanical elements of play, as opposed to the visual ones.

Whether your computer has 'arcade quality' graphics or not, you will need a basic understanding of coordinate geometry, which in simple terms, means working with ARRAY(x,y).

In coordinate geometry the x value is referred to as the x-axis, the y value as the y-axis. The benefit of using this system are paramount because we are able to keep track of all 'objects' and their relationship with one another. Let's say we are using a linear array, and trying to get an 'alien' to 'crash' into the player's spaceship. The VDU RAM we are using is 64 columns by 16 rows (1K) starting at 15360, the alien is at address 15999, and the player at 16043. On the VDU the alien will appear above and to the right of the player.

How can the programmer deduce this from the numbers 15999 and 16043? With a great deal of difficulty. If we simply moved the 'alien' closer by adding '1', the alien would appear to move further away – in the opposite direction! Or at least move further away before getting closer. To partially compensate for this problem, we can subtract the player's location from the alien's to see if it is further than 64 bytes away. If so, the alien can leap after the player's spaceship in sets of 64. Nevertheless, the 'alien' at times will still appear to move in the opposite direction.

Had we converted the numbers 15999 and 16043 to a two dimensional array (treating all VDU RAM as one big data table) we would have locations PLAYER(20,11) and ALIEN(63,10). That is, 20 columns from the right by 11 rows down, and, 63 columns from the right by 10 rows down. The alien can track the player by comparing the x and y-axis. Now the programmer can deduce that the player's craft is 43 bytes to the left, and one column (64 bytes) below.

Using coordinate geometry it's possible to work with formulas to find the distance between two objects, mid-points, ratios and gradients to name a few. It is also possible to draw a line between two points, or construct a circle or parabola. Really however, although occasionally used in arcade games, many of these formulas require the use of fractions, and functions like square roots. Should you be interested in these concepts for your own programs, an elementary maths textbook usually outlines most of these principles. Be warned, some of your routines will need to be very sophisticated, unless you use special 'scaling' techniques (which is a complex topic in itself). This chapter is only basic however, so you must refer to other sources if you should wish to apply them.

The more or less 'standard' two dimensional array to linear array conversion routine used in this chapter, is as follows:

```
;CALCULATE VDU COORDINATES
;ENTRY -->    DE = ARRAY(X,Y)
;EXIT  -->    IY = ADDRESS
VDU     EQU   15360        ;START VDU RAM
COLUMN  EQU   64           ;COLUMN LENGTH
GETL    PUSH  BC
        LD    B,E
        LD    IY,VDU-COLUMN
        EX    DE,HL
        LD    DE,COLUMN
GLP     ADD   IY,DE
        DJNZ  GLP
        LD    C,H
        LD    B,0
        ADD   IY,BC
        EX    DE,HL
        POP   BC
        RET
```

The register-pair DE holds the ARRAY(*x*,*y*) coordinates, and on exit IY holds the address the array is pointing to. Register E holds the *y*-axis, D the *x*-axis.

Using this subroutine we can begin our first experiments with animation.

To animate a graphic block we must display it, move it horizontally or vertically alongside itself, and then remove the previous graphic block. You might probably have experimented with this technique in a high level language. Examine listing 5.1, which could be the basis for a video tennis game.

The program simulates a 'bouncing ball', with the ball represented as a moving byte. The ball moves diagonally across the screen, bouncing in the opposite direction when 'hitting' the edge of the screen.

The VDU is cleared using a block move, DE is loaded with the *x* and *y*-axis appropriately, and HL is loaded with the increments. Register H is loaded with one, and L with 255, which serves the purpose of '-1' when added to the D or E register. The DE register pair is then adjusted according to the contents of HL. The new contents of DE are converted back into a linear array using the GETL routine, and a white graphic block, byte '191', is loaded into the corresponding address.

After this, D and E are checked to see if they have reached the edge of the screen. The maximum and minimum *x*-axis being zero and 63, the maximum and minimum *y*-axis being one and 16 *(the y-axis should always start at one, rather than zero,* else the DJNZ instruction cause 64 to be added to IY 255 times). If the graphic block is at an address that borders the edge of the VDU, the contents of H or L are negated (255 is subtracted) to switch the '1' to a '-1' or vice versa. Consequently, the byte seems to bounce off the edge of the screen when the addition or subtraction is reversed.

The 'ball' or graphic block or 'alien' is loaded directly into VDU RAM using a LD (IY),nn instruction, where nn is hopefully a suitable byte from your computer's character set. Again let me emphasize that this method is valid with most computers – but not all. If you are unsure whether or not this is valid for your system, refer to your technical or BASIC (if one is supplied) user manual. Although there is computer reference material supplied in the appendixes in this book, Murphy's Law firmly dictates that your system will probably not be mentioned.

What was once a simple LD instruction might have to be replaced with special machine language subroutines and system software (subroutines in your ROM). If your reference material makes mention of things like 'display files' and 'attribute files' when discussing VDU RAM, you probably have such a VDU layout. If you cannot get any technical information about your computer (believe it or not, this happens) you will have to write a disassembler (see chapter seven) and study your ROM. (You might be wondering how you can write a disassembler if you have no technical information: do it in a high level language.) This contains all the technical information you will need to know, provided you have the perseverance to decipher completely undocumented machine language.

```
                00100 ;ANIMATION DEMONSTRATION
                00110 ;*CHANGE ACCORDING TO COMPUTER
                00120 ;*****************************
                00130 ;
3C00            00140 VDU       EQU     15360           ;*START OF VDU RAM
03FF            00150 LEN       EQU     1023            ;*LENGTH OF VDU RAM-1
0010            00160 ROW       EQU     16              ;*NUMBER OF ROWS
0040            00170 COLUMN    EQU     64              ;*NUMBER OF COLUMNS
0020            00180 BLANK     EQU     32              ;*BLANK CHARACTER
00BF            00190 WHITE     EQU     191             ;*ALL WHITE GRAPHIC BLOCK
9000            00200           ORG     9000H           ;*
9000 F3         00210 START     DI
9001 31FFFF     00220           LD      SP,65535        ;*
9004 21003C     00230           LD      HL,VDU          ;CLEAR VDU RAM
9007 11013C     00240           LD      DE,VDU+1
900A 01FF03     00250           LD      BC,LEN
900D 3620       00260           LD      (HL),BLANK
900F EDB0       00270           LDIR
9011 110A20     00280           LD      DE,200AH        ;*ARRAY(32,10)
9014 21FF01     00290           LD      HL,01FFH        ;*DELTA 1 & -1
9017 FD21003C   00300           LD      IY,VDU
901B 018813     00310 LOOP      LD      BC,5000         ;*DELAY LOOP
901E 0B         00320 LOOP1     DEC     BC
901F 78         00330           LD      A,B
9020 B1         00340           OR      C
9021 20FB       00350           JR      NZ,LOOP1
9023 FD360020   00360           LD      (IY),BLANK
9027 7A         00370           LD      A,D             ;ADD INC,
9028 84         00380           ADD     A,H             ;TO ARRAY
9029 57         00390           LD      D,A
902A 7B         00400           LD      A,E
902B 85         00410           ADD     A,L
902C 5F         00420           LD      E,A
902D D5         00430           PUSH    DE
902E E5         00440           PUSH    HL
                00450 ;
902F 43         00460           LD      B,E             ;GETL ROUTINE HERE
9030 FD21C03B   00470           LD      IY,VDU-COLUMN
9034 EB         00480           EX      DE,HL
9035 114000     00490           LD      DE,COLUMN
9038 FD19       00500 GLP       ADD     IY,DE
903A 10FC       00510           DJNZ    GLP
903C 4C         00520           LD      C,H
903D 0600       00530           LD      B,0
903F FD09       00540           ADD     IY,BC
9041 EB         00550           EX      DE,HL
9042 E1         00560           POP     HL
9043 D1         00570           POP     DE
                00580 ;
9044 FD3600BF   00590           LD      (IY),WHITE
9048 7A         00600           LD      A,D
9049 FE3F       00610           CP      COLUMN-1        ;HIT EDGE?
904B 2803       00620           JR      Z,FIX
904D 3D         00630           DEC     A               ;HIT EDGE?
904E 2004       00640           JR      NZ,CONT
9050 7C         00650 FIX       LD      A,H
9051 ED44       00660           NEG                     ;REVERSE DIRECTION
9053 67         00670           LD      H,A
9054 7B         00680 CONT      LD      A,E
9055 3D         00690           DEC     A               ;HIT EDGE?
9056 2804       00700           JR      Z,FIX1
9058 FE10       00710           CP      ROW             ;HIT EDGE?
905A 20BF       00720           JR      NZ,LOOP
905C 7D         00730 FIX1      LD      A,L
905D ED44       00740           NEG                     ;REVERSE DIRECTION
905F 6F         00750           LD      L,A
9060 18B9       00760           JR      LOOP
9000            00770           END     START
00000 Total errors
```

5.4 ARCADE GAMES

Arcade or 'video' games are ideal for programmers looking to better their Z80 technique. A well *structured, modular* programming approach is not only recommended, but essential. A game of this nature has to keep track of the player's spacecraft, aliens, scenery, player's and alien's weapons, background sounds, scores, and fuel levels, etc., all in relation to one another and all at once. Every subroutine has to be designed to do something very small each time it is CALLed, whether this is moving an asteroid fractionally left, or a 'bomb' downwards minutely.

There are mainly two types of information the programmer will be dealing with; conventional data for alien positions, shape, attack patterns; stored away in arrays and tables, and counters; which keep the timing of the game right, stored in individual addresses. In other words, if you have avoided index-registers before, because they are lengthy and slower than other instructions, you will begin to see them in a new light from here on.

Our first arcade game is a conventional one: player's spaceship or 'laser cannon' at the bottom of the screen, with mobility restricted to moving left or right, marauding aliens appearing at the top, and moving down. Their means of attack, besides ramming, is dropping bombs; the player's, a shot fired from the 'laser cannon'. Different types of 'aliens' come and go in 'attack wave' fashion.

This straightforward game can be broken down into eight subroutines, as shown in diagram 1.1, a runchart of the game.

The programmer should try to work from a 'clean slate' by removing all 'junk' from VDU RAM left over from the previous frame. Either block move the VDU with a blank graphic block or use a graphic block ideal for a background colour.

The order of operation is very important and should be well thought out before beginning. While minor changes are permissible, most subroutines depend on others doing their job first. This will become very clear as we progress.

If you thought the most logical first subroutine would be to display the player's 'laser cannon', you would be wrong. This is one of the last subroutines to be executed. What we want to know first is where the player intends moving and, if possible, whether or not to fire. A scan of the keyboard is required.

The label LOC is the address of the player's 'laser cannon' (in this example, there is no need to store the player's location in *array* form since the aliens are not concerned where it is). The label KEY is the memory location used to check the keyboard (whatever this may be on your system):

```
INP     LD      HL,(LOC)
        LD      A,(KEY)
        BIT     6,A
        JR      NZ,RGHT
        BIT     5,A
        JR      NZ,LFT
        RET
```

The instructions BIT 6,A and BIT 5,A are typical examples. When a '1' appears in the sixth or seventh bit position of the byte KEY, (the direction keys are being pressed) a jump is made to either label RGHT, short for 'move laser cannon right', or LFT, short for 'move laser cannon left'. The corresponding address of KEY, the bits used, and the JR NZ (possibly Z) will vary according to your computer's requirements.

The routine LFT decrements the contents of HL, moving the location of the player's craft one byte to the left, and then checks if the edge of the screen has been reached:

```
LFT     DEC     HL
        LD      A,L
        CP      192
        RET     Z
        LD      (LOC),HL
        RET
```

If the compare *is* invalid, and the player's 'laser cannon' is bordering on the edge of the VDU, the new location is not stored in LOC. In effect the move is cancelled.

What is the particular significance of the number 192? The same 1K of VDU RAM we have used so far, that is: 64 columns across, starting at address 15360

and ending at 16383, has addresses between 16320 and 16383 (the last column) representing the bottom line. In hexadecimal these numbers are 3FC0H and 3FFFH. Register L loaded with the LSD of the former address will hold 0C0H (192), or 0FFH (255) if the latter.

So, we know that the L register cannot equal these two values without the risk of the player's craft running off the bottom line. By checking the L register we keep the spaceship where we want it.

The player still has one more option, the firing of a laser, but we will check for that later. Now we have to turn to creating the aliens.

Each 'alien' is a set of data containing information about its type (how else are we going to tell if they're Grodilixxians or Arrrgorians?), whether they are 'on' (attacking) or 'off' (destroyed), their next move and their current position. This information can be contained in five bytes, so, not surprisingly, they are called 'five byte aliens'.

Some 'aliens' need only two bytes – to keep track of their current position, with other information held in a separate table. Others can be eight bytes or more when it becomes necessary to keep count of the number of hits each alien has received (for games where one hit is not enough to destroy them), and more sophisticated aspects like strategy. For now we'll concentrate on the basic 'alien'.

In the following routine two labels are used; ALEFT which keeps track of the number of 'aliens' on the VDU at any one time, and the counter TYPE, which systematically determines which type of 'alien' is going to attack next:

```
CALIEN  LD      A,(ALEFT)
        OR      A               ;ALIENS GONE?
        RET     NZ              ;NO, RETURN
        LD      A,5             ;NO. OF ALIENS,
        LD      (ALEFT),A       ;ON SCREEN
        LD      B,5             ;NO. OF ALIENS
        LD      IX,ALIEN        ;ALIEN TABLE
        LD      D,1             ;Y-AXIS
```

```
        LD      E,5             ;X-AXIS
        LD      A,(TYPE)        ;ATTACK WAVE
        INC     A               ;SELECT NEXT
        CP      4               ;NO SUCH WAVE!
        JR      Z,FIX           ;FIX PROBLEM
        LD      (TYPE),A        ;LAST WAVE
        CALL    PAT             ;HL=ATTACK P.
CLP     LD      A,(TYPE)        ;ATTACK WAVE
        LD      (IX),A          ;STORE AWAY
        LD      (IX+3),L        ;STORE THE,
        LD      (IX+4),H        ;PATTERN
        LD      (IX+1),D        ;STORE THE,
        LD      (IX+2),E        ;X,Y-AXIS
        LD      A,E
        LD      C,11            ;MOVE ALIENS,
        ADD     A,C             ;APART 11,
        LD      E,A             ;BYTES
        PUSH    DE
        LD      DE,5
        ADD     IX,DE           ;NEXT ALIEN
        POP     DE
        DJNZ    CLP             ;ALL 5 DONE?
```

```
        RET
FIX     LD      A,1             ;FIRST WAVE
        RET
PAT     DEC     A               ;JUMP TO,
        JR      Z,P1            ;THE CORRECT,
        DEC     A               ;ATTACK PATTERN
        JR      Z,P2
        LD      HL,PAT3         ;PATTERN 3
        RET
P1      LD      HL,PAT1         ;PATTERN 1
        RET
P2      LD      HL,PAT2         ;PATTERN 2
        RET

ALEFT   DB      0
TYPE    DB      0
PAT1    DB      255,4,2,2,2,254,5,3,3,3,1
PAT2    DB      255,1
PAT3    DB      255,1,2,2,254,1,3,3,1
        DB      255
```

Let us examine this routine step by step. The accumulator is loaded with ALEFT, which is the number of 'aliens' still attacking (those that have not yet been destroyed). If there are none left, the player has been successful in destroying them all, or they have reached the bottom of the screen and have 'turned themselves off' (more on this later). Unlike other games, where aliens keep coming constantly, further 'aliens' will not appear until all are gone.

The routine therefore sets out to restore all the 'aliens' at once. First, ALEFT is reloaded with five, the number of 'aliens' to appear on the screen, and IX is loaded with ALIEN, the first address of the table that holds all the 'alien' data. The DE register pair is loaded with appropriate x and y-axis coordinates up near the top of the screen. The accumulator is loaded with the contents of TYPE, which keeps track of the 'aliens' shape and movements.

The accumulator is then incremented to point to the new type. If this equals '4', FIX is called and the type reverts to '1', since there are only three types of 'aliens' in this example. The routine PAT is CALLed, which points HL to the correct 'attack pattern' according to alien type. The 'attack pattern' determines the 'aliens' moves and when to drop bombs.

On returning HL, DE and A are saved away in the ALIEN table. Then, the x-axis is adjusted so that the next 'alien' will appear to the right of the last one (there is little point in having five 'aliens' if you put them all in the same location). Finally, five bytes are added to IX and the routine repeats until all 'aliens' are newly initialised. The three labels; PAT1, PAT2 and PAT3 contain the data used in determining the next move.

Now that we have given our 'aliens' a location, a pattern and a classification, we can display them on the VDU.

To display the 'aliens' on the VDU, a routine must get their current location, and put graphic blocks that make up their shapes in these addresses. The character set of the computer – especially graphics – vary significantly, and there might even be a high and low resolution graphics mode. The one choosen is a set of all possible combinations of a six pixel to the byte character set. See Diagram 5.4.

Try designing 'aliens' and other objects by drawing up grids equivalent to your own computer's VDU layout (you may be able to buy this paper from where you purchased your computer), and fill in the graphic blocks according to taste. You may have to plot your shapes using colour pens if your system has colour. For the demonstration we are not going to worry about that.

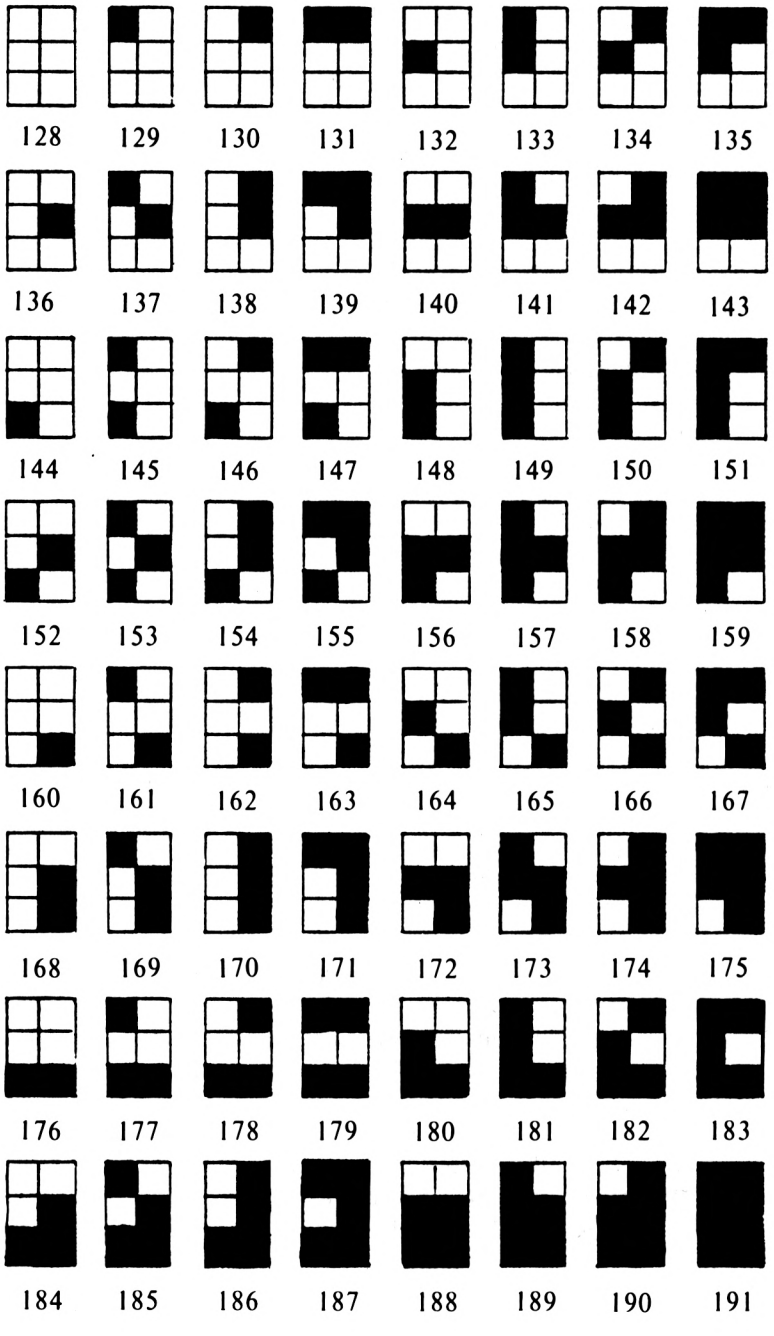
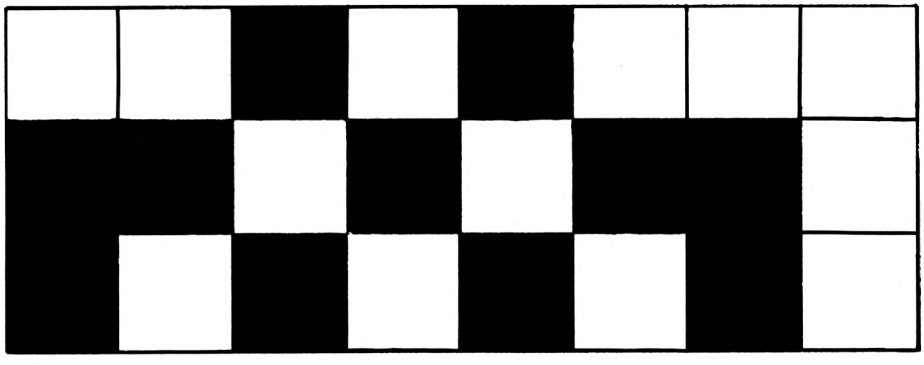


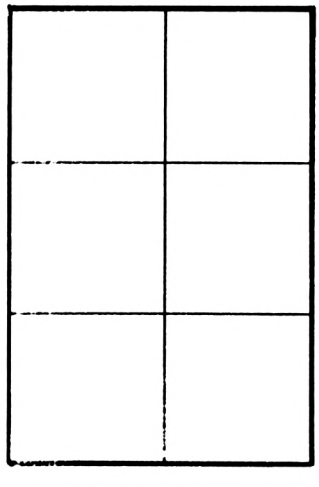
DIAGRAM 5.4 – TYPICAL GRAPHIC BLOCK CHARACTER SET

Diagram 5.5, which is a set of graphic blocks brought together to represent our 'alien' – well, since we are using only one K of RAM without colour, try and use your imagination . . .



FIRST BYTE SECOND BYTE THIRD BYTE FOURTH BYTE

DIAGRAM 5.5 – GRAPHIC BLOCK "ALIEN" DESIGN



GRAPHIC BLOCK

The following routine looks up the array location in the ALIEN table, converts it to an address, and displays the graphics on the screen:

```
PALIEN LD     IX,ALIEN    ;DATA TABLE
       LD     B,5         ;NO. OF ALIENS
PLP    LD     A,(IX)      ;GET TYPE
       OR     A           ;IS IT,
       JR     Z,FLIP      ;DESTROYED?
       LD     D,(IX+1)    ;Y-AXIS
       LD     E,(IX+2)    ;X-AXIS
       CALL   GETL        ;IY=ADDRESS
       LD     A,(IX)      ;WHAT SHAPE?
       DEC    A           ;TYPE 1
       JR     Z,A1
       DEC    A           ;TYPE 2
       JR     Z,A2
       JR     A3          ;TYPE 3
FLIP   LD     DE,5        ;NEXT ALIEN
       ADD    IX,DE
       DJNZ   PLP         ;ALL DONE?
       RET                ;YES
```

The routine GETL, as you should recall, converts the coordinates into a specific memory location. When an 'alien' is destroyed, its TYPE reverts to zero, so it is not redisplayed on the VDU. The routines A1, A2 and A3 load the graphic blocks into the corresponding memory addresses – held in IY:

```
;ALIEN SHAPE IN 3 GRAPHIC BLOCKS
A1      LD      (IY),157
        LD      (IY+1),179
        LD      (IY+2),174
        JR      FLIP
        .   .   .
```

The five 'aliens' are now displayed on the VDU. Of course, graphics change so radically on each computer system that you may have to do something completely different to produce the same effect.

The routine FIRE checks if the player wants to fire, or, if already fired, keeps track of its location and if it has hit anything. The first part of the FIRE routine looks up COUNT, which is zero if the 'laser' has already been fired, and loads the HL register pair with the location of the player's 'laser cannon'. If zero (no shot on screen), it checks if the player wants to fire:

```
FIRE    LD      A,(COUNT)   ;LASER BEAM
        LD      HL,(LOC)    ;PLAYER'S LOC
        OR      A           ;ANYTHING FIRED?
        JR      NZ,FIRE2
        LD      A,(KEY)     ;CHECK KEYBOARD
        BIT     7,A         ;FIRE KEY PRESS?
        RET     Z           ;NO FIRING
        INC     HL          ;ADJUST LASER
        LD      A,14        ;MAX. MOVE UP
        LD      (COUNT),A
        LD      (FLOC),HL   ;FIRE LOCATION
```

Again, BIT 7,A is only an example; it could be any bit depending on the key selected. It could even be a IN A,(n) instruction if using the joystick (refer to your own technical information). The location of the player's 'laser' is saved in FLOC. The 'laser' beam moves straight up the screen (a total of 16 rows), starting at row 15 and 'turning off' (if the laser does not hit anything) at row '1'. The number 14 is therefore placed in COUNT to keep track of the 'laser's' position. The program follows through:

```
FIRE2   LD      HL,(FLOC)
        LD      A,(COUNT)
        LD      B,A             ;COUNTER
        LD      DE,-64          ;COLUMN LENGTH
FLP0    ADD     HL,DE           ;MOVE UP 1 ROW
        LD      A,(HL)
        CP      128             ;BLANK SPACE?
        JR      NZ,SKIP
        LD      (HL),149        ;SHAPE OF LASER
        DEC     B               ;1 LESS ROW
        LD      A,B
        LD      (COUNT),A       ;SAVE IT
        LD      (FLOC),HL       ;NEW LOC
        RET
SKIP    LD      (HL),0          ;HIT ALIEN!
        XOR     A               ;END OF LASER,
        LD      (COUNT),A       ;SHOT
        RET
```

The routine FIRE2 moves the 'laser' one row at a time up the screen. Before placing the 'laser' graphic block in the new location, a check is made for 128. This is the background graphic block, hence, anything else must be a piece of 'alien'. If the 'laser' has struck something, a jump is made to SKIP, which 'turns off' the 'laser' and places a zero byte in the address. Now, another routine has to check each 'alien' location for a zero byte. If it finds one, the 'alien' is 'turned off' as well.

The next routine handles bomb dropping. For the first few times it is executed, nothing happens as the BULLET subroutine is activated by another subroutine we have not yet reached. So we will skip this one, and refer back to it later.

The player, having fired his 'laser', might have been successful in blasting the enemy. The following routine checks for the zero byte that would indicate this:

```
VALIEN LD     IX,ALIEN    ;VERIFY ALIEN
       LD     B,5         ;NO. ALIENS
       LD     A,(IX)
       OR     A           ;ALREADY,
       JR     Z,PASS      ;DESTROYED?
       LD     D,(IX+1)    ;Y-AXIS
       LD     E,(IX+2)    ;X-AXIS
       CALL   GETL        ;IY=ADDRESS
       XOR    A           ;CHECK Ø
       CP     (IY)        ;1ST BYTE
       JR     Z,EX        ;EXPLOSION
       CP     (IY+1)      ;2ND BYTE
       JR     Z,EX
       CP     (IY+2)      ;3RD BYTE
       JR     Z,EX
```

```
PASS    LD      DE,5            ;NEXT ALIEN
        ADD     IX,DE
        DJNZ    VLP
        RET
```

The three compares check each byte of the 'alien' for zero. If destroyed, a jump is made to EX:

```
EX      LD      (IY),191        ;ALL WHITE,
        LD      (IY+1),191      ;GRAPHIC BLOCK
        LD      (IY+2),191
        LD      (IX),0          ;TURN OFF
        LD      A,(ALEFT)
        DEC     A
        LD      (ALEFT),A
        JR      PASS
```

The explosion is a modest one, with just three all-white graphic blocks replacing the previous shape. A fantastic particle explosion will be left up to the reader as an additional exercise.

Accordingly, the 'alien' is 'turned off' with its TYPE identifier loaded with zero, and the ALEFT counter is adjusted to indicate one less on the screen.

A very similar routine is used to check and display the player's 'laser cannon'. No Z80 code is even needed to describe the routine. Just load an index register with the contents of LOC, the accumulator with 128 (the blank space or possibly the background colour) and check that all three bytes or so are clear. If not, the player's craft has been hit by an 'alien' or one of its 'bombs'. Otherwise display the 'laser cannon' in a similar fashion to that of the 'aliens'.

The subroutine WALIEN, short for 'Work out Alien's next move' is the most sophisticated one in the program. It needs to know where the 'alien' is currently

located, and what data item must be processed. Initially all registers are loaded with the necessary information:

```
WALIEN LD     B,5           ;NO. ALIENS
       LD     IX,ALIEN
       LD     A,(IX)
       OR     A
       JR     Z,NEXT
       LD     D,(IX+1)      ;Y-AXIS
       LD     E,(IX+2)      ;X-AXIS
       LD     H,(IX+4)      ;ADDRESS
       LD     L,(IX+3)      ;ADDRESS
       CALL   MOVE          ;NEXT MOVE
       LD     (IX+1),D
       LD     (IX+2),E
       LD     (IX+4),H
       LD     (IX+3),L
NEXT   LD     DE,5
       ADD    IX,DE
       DJNZ   WLP
       RET
```

Which should be fairly straightforward to you by now. The routine MOVE does all the real work:

```
MOVE    INC     HL              ;NEXT DATA
        LD      A,(HL)          ;READ IT
        CP      255             ;TERMINATOR
        JR      Z,RESET
        CP      254             ;DROP BOMB
        CALL    Z,LASER
        .   .   .
```

The HL register pair is adjusted to point to a new piece of data. This has two special codes; 255 acts as a terminator, telling the program that the end of the 'attack pattern' has been reached, so the routine RESET can readjust HL to point to the beginning again (the pattern repeats indefinitely); and 254 instructs the program to drop a 'bomb' on the player, which is handled by routine LASER.

The RESET routine pushes the data pointer (contents of HL) back in memory until 255 is reached. That being the first byte of the 'attack pattern'.

```
RESET   DEC     HL
        LD      A,(HL)
        INC     A
        JR      Z,MOVE
        JR      RESET
```

Besides 255 and 254, MOVE also recognises the numbers 1 through 8, which correspond to moves; north, south, east, west, south-east, south-west, north-west and north-east respectively. The next routine will analyse each number and adjust the E or D register holding the *x,y*-axis accordingly. For example: to move north-east the program must DEC D (*y*-axis in this example) and INC E (*x*-axis in this example). Examine the following routine:

```
        LD      IY,MTAB         ;MOVEMENT TABLE
M1      DEC     A
```

```
        OR      A
        JR      Z,READ          ;GOT IT!
        INC     IY              ;NEXT INSTRUCT
        INC     IY              ;   "      "
        JR      M1
READ    LD      A,(IY)          ;GET INSTRUCT
        LD      (INST),A        ;PUT IN "INST"
        LD      A,(IY+1)        ;GET NEXT
        LD      (INST+1),A
INST    NOP                     ;INSTRUCT HERE
        NOP                     ;AND HERE
        LD      A,D             ;AT BOTTOM OF,
        CP      17              ;OF VDU RAM?
        RET     NZ
        LD      (IX),0          ;THEN "TURN OFF"
        LD      A,(ALEFT)
        DEC     A
        LD      (ALEFT),A
        RET
MTAB    DW      1500H,1400H,1C00H,1D00H
        DW      1C14H,141CH,1D15H,151CH
```

The table MTAB holds the Z80 instruction code for incrementing and decrementing the D and E registers. Had the data in the current 'attack pattern' been a '2' for example, IY would be adjusted to point to the second word in the table – that of 1400H. This would be loaded into INST and the next byte. When INST is executed, the Z80 reads INC D and NOP, which will change the *y*-axis.

A check is made to see that the new *x,y*-axis values do not exceed the permissible VDU limits, in which case the 'alien' is 'turned off' (by loading the first byte pointed to by IX with zero).

A question: If all five 'aliens' read the same data table, and make five identical moves, why store five separate data pointers away? That is fine if you want all the 'aliens' to move in circles together, and to the right together and to the left together and so on, as they do in this program. However, the routine CALIEN is only an example. One which happens to 'turn on' all the 'aliens' at the same time. Another routine could be written that 'turns on' another 'alien' as soon as one is destroyed. In such a situation all the 'aliens' would be at different stages in the 'attack pattern' and therefore need separate data table pointers.

The LASER and BULLET routines act as small CALIEN and WALIEN routines, but keep track of the simpler enemy bombs, which, gratefully, move in only one direction – down.

There is a maximum limit of five bombs moving at once, one for each 'alien'. Since CALIEN has all the 'alien' movements synchronised, all 'bombs' are launched simultaneously. A separate data table called BTAB keeps track of their location. Such a routine would look something like the following:

```
LASER   INC     HL              ;NEXT DATA ITEM
        PUSH    BC
        LD      IY,BTAB
        LD      B,5
BLP     LD      A,(IY)
        OR      A
        JR      NZ,AGAIN        ;IS IT ON?
```

```
        LD      A,D             ;PUT BOMB,
        INC     A               ;UNDER ALIEN
        LD      (IY+1),A        ;SAVE LOCATION
        LD      (IY+2),E        ;   "      "
        LD      (IY),1          ;BOMB "ON"
        JR      AGAIN1
AGAIN   INC     IY              ;NEXT ALIEN
        INC     IY
        INC     IY
        DJNZ    BLP
AGAIN1  POP     BC
        LD      A,(HL)          ;NEXT MOVE
        RET
```

With all the data placed in BTAB we can finalize the program with a routine to keep the bombs 'falling':

```
BULLET  LD      IX,BTAB
        LD      B,5
BLP2    LD      A,(IX)
        OR      A
        JR      Z,TRY
        LD      D,(IX+1)        ;GET LOCATION
        LD      E,(IX+2)
```

```
        INC     D               ;BOMB 'FALLS'
        LD      A,D             ;CHECK END
        CP      17              ;AT BOTTOM?
        JR      Z,BOFF
        LD      (IX+1),D        ;SAVE IT
        CALL    GETL            ;IY=ADDRESS
        LD      (IY),136        ;GRAPHIC BLOCK
TRY     INC     IX              ;NEXT BOMB
        INC     IX
        INC     IX
        DJNZ    BLP2
        RET
BOFF    LD      (IX),0          ;BOMB "OFF"
        JR      TRY             ;TRY NEXT
```

The routine increments the *y*-axis (register D here) and displays the graphic block – representing the bomb – on the screen. Now all items that are going to be displayed have been so. The program, in its present form, would run far too fast on even the slowest Z80, so an appropriate delay loop of between 10000 and 50000 superfluous instructions should be executed before the screen is 'cleaned' again by a block move and everything is redisplayed.

Listing 5.2 is a complete and working 'video game' brought together by the routines covered here. Although it is only 782 bytes in length, it does do a great deal. The reader should go over the listing carefully, and then refer back to the various references made in this section. As you will see, it is more of a compromise between complexity and simplicity. By rewriting the whole program, it is possible to use a lot less memory, but this might seriously specialize the program. In its present form, it can be modified to handle more sophisticated scenarios as well.

```
                00100 ;VIDEO GAME
                00110 ;*CHANGE ACCORDING TO COMPUTER
                00120 ;*****************************
                00130 ;
3C00            00140 VDU       EQU     15360           ;*START OF VDU RAM
3840            00150 KEY       EQU     14400           ;*MEMORY-MAPPED KEY
03FF            00160 LEN       EQU     1023            ;*LENGTH OF VDU
0040            00170 COLUMN    EQU     64              ;*COLUMN LENGTH
0080            00180 BLANK     EQU     128             ;*BLANK GRAPHIC BLOCK
00BF            00190 WHITE     EQU     191             ;*ALL WHITE GRAPHIC BLOCK
0BB8            00200 DELAY     EQU     3000            ;*DELAY BETWEEN FRAMES
9000            00210           ORG     9000H           ;*
9000 F3         00220 START     DI
9001 21C03F     00230           LD      HL,VDU+LEN-COLUMN+1
9004 22F192     00240           LD      (LOC),HL        ;PLAYER'S SHIP
9007 3E05       00250           LD      A,5             ;NO. OF SHIPS
9009 32F392     00260           LD      (ME),A
900C 3E03       00270           LD      A,3             ;ALIEN TYPE
900E 32F492     00280           LD      (TYPE),A
9011 CDA692     00290           CALL    PAUSE
9014 21003C     00300 RUN       LD      HL,VDU          ;EXECUTION CONTROL LOOP
9017 11013C     00310           LD      DE,VDU+1
901A 01FF03     00320           LD      BC,LEN
901D 3680       00330           LD      (HL),BLANK
901F EDB0       00340           LDIR                    ;CLEAR VDU
9021 31FFFF     00350           LD      SP,65535        ;*
9024 CD7790     00360           CALL    INP             ;CHECK KEYBOARD
9027 CD3692     00370           CALL    CALIEN          ;CHECK ALIEN
902A CD6D91     00380           CALL    PALIEN          ;DISPLAY ALIEN
902D CD9890     00390           CALL    FIRE            ;FIRE LASER
9030 CD0592     00400           CALL    BULLET          ;ALIEN BULLETS
9033 CDBF91     00410           CALL    VALIEN          ;ALIEN HIT?
9036 CD4690     00420           CALL    OK              ;PLAYER HIT?
9039 CDD290     00430           CALL    WALIEN          ;GET NEXT ALIEN MOVE
903C 01B80B     00440           LD      BC,DELAY        ;SLOW GAME DOWN
903F 0B         00450 LP0       DEC     BC
9040 78         00460           LD      A,B
9041 B1         00470           OR      C
9042 20FB       00480           JR      NZ,LP0
9044 18CE       00490           JR      RUN             ;AGAIN
                00500 ;
9046 DD2AF192   00510 OK        LD      IX,(LOC)        ;CHECK IF PLAYER HIT
904A 3E80       00520           LD      A,BLANK         ;*
904C DDBE00     00530           CP      (IX)
904F 2017       00540           JR      NZ,DEAD
9051 DDBE01     00550           CP      (IX+1)
9054 2012       00560           JR      NZ,DEAD
9056 DDBE02     00570           CP      (IX+2)
9059 200D       00580           JR      NZ,DEAD
905B DD3600B8   00590           LD      (IX),184        ;*SUITABLE GRAPHIC,
905F DD3601BD   00600           LD      (IX+1),189      ;*BLOCKS SHOULD BE,
9063 DD360290   00610           LD      (IX+2),144      ;*PLACED HERE
9067 C9         00620           RET
9068 3AF392     00630 DEAD      LD      A,(ME)
906B 3D         00640           DEC     A
906C B7         00650           OR      A
906D 2891       00660           JR      Z,START
906F 32F392     00670           LD      (ME),A
9072 CDA692     00680           CALL    PAUSE
9075 189D       00690           JR      RUN
                00700 ;
9077 2AF192     00710 INP       LD      HL,(LOC)        ;PLAYER'S MOVE
907A 3A4038     00720           LD      A,(KEY)
907D CB77       00730           BIT     6,A             ;*
907F 2005       00740           JR      NZ,RGHT
9081 CB6F       00750           BIT     5,A             ;*
9083 200A       00760           JR      NZ,LFT
9085 C9         00770           RET
9086 23         00780 RGHT      INC     HL
9087 7D         00790           LD      A,L
9088 FEFE       00800           CP      254             ;*
```

```
908A C8          00810          RET     Z
908B 22F192      00820          LD      (LOC),HL
908E C9          00830          RET
908F 2B          00840 LFT      DEC     HL
9090 7D          00850          LD      A,L
9091 FEC0        00860          CP      192             ;*
9093 C8          00870          RET     Z
9094 22F192      00880          LD      (LOC),HL
9097 C9          00890          RET
                 00900 ;
9098 3AF892      00910 FIRE     LD      A,(COUNT)       ;PLAYER'S SHOT
909B 2AF192      00920          LD      HL,(LOC)
909E B7          00930          OR      A
909F 200F        00940          JR      NZ,FIRE2
90A1 3A4038      00950          LD      A,(KEY)
90A4 CB7F        00960          BIT     7,A             ;*
90A6 C8          00970          RET     Z
90A7 23          00980          INC     HL
90A8 3E0E        00990          LD      A,14            ;*PLAYER'S LASER LIMIT
90AA 32F892      01000          LD      (COUNT),A
90AD 22F692      01010          LD      (FLOC),HL
90B0 2AF692      01020 FIRE2    LD      HL,(FLOC)
90B3 3AF892      01030          LD      A,(COUNT)
90B6 47          01040          LD      B,A
90B7 11C0FF      01050          LD      DE,-COLUMN
90BA 19          01060 FLP0     ADD     HL,DE
90BB 7E          01070          LD      A,(HL)
90BC FE80        01080          CP      BLANK
90BE 200B        01090          JR      NZ,SKIP
90C0 3695        01100          LD      (HL),149        ;*GRAPHIC BLOCK
90C2 05          01110          DEC     B
90C3 78          01120          LD      A,B
90C4 32F892      01130          LD      (COUNT),A
90C7 22F692      01140          LD      (FLOC),HL
90CA C9          01150          RET
90CB 3600        01160 SKIP     LD      (HL),0
90CD AF          01170          XOR     A
90CE 32F892      01180          LD      (COUNT),A
90D1 C9          01190          RET
                 01200 ;
90D2 0605        01210 WALIEN   LD      B,5             ;ALIEN'S NEXT MOVE
90D4 DD21C492    01220          LD      IX,ALIEN
90D8 DD7E00      01230 WLP      LD      A,(IX)
90DB B7          01240          OR      A
90DC 2812        01250          JR      Z,NEXT
90DE DD5601      01260          LD      D,(IX+1)
90E1 DD5E02      01270          LD      E,(IX+2)
90E4 DD6604      01280          LD      H,(IX+4)
90E7 DD6E03      01290          LD      L,(IX+3)
90EA CDF890      01300          CALL    MOVE
90ED CD9992      01310          CALL    LOAD
90F0 110500      01320 NEXT     LD      DE,5
90F3 DD19        01330          ADD     IX,DE
90F5 10E1        01340          DJNZ    WLP
90F7 C9          01350          RET
90F8 23          01360 MOVE     INC     HL
90F9 7E          01370          LD      A,(HL)
90FA FEFF        01380          CP      255
90FC 2841        01390          JR      Z,RESET
90FE FEFE        01400          CP      254
9100 CC4691      01410          CALL    Z,LASER
9103 FD212F91    01420          LD      IY,MTAB
9107 3D          01430 M1       DEC     A
9108 B7          01440          OR      A
9109 2806        01450          JR      Z,READ
910B FD23        01460          INC     IY
910D FD23        01470          INC     IY
910F 18F6        01480          JR      M1
9111 FD7E00      01490 READ     LD      A,(IY)
9114 321D91      01500          LD      (INST),A
9117 FD7E01      01510          LD      A,(IY+1)
```

```
911A 321E91    01520          LD      (INST+1),A
911D 0000      01530 INST     DW      0
911F 7A        01540          LD      A,D
9120 FE11      01550          CP      17              ;*ALIEN LIMIT
9122 C0        01560          RET     NZ
9123 DD360000  01570          LD      (IX),0
9127 3AF592    01580          LD      A,(ALEFT)
912A 3D        01590          DEC     A
912B 32F592    01600          LD      (ALEFT),A
912E C9        01610          RET
912F 0015      01620 MTAB     DW      1500H,1400H,1C00H,1D00H
     0014 001C 001D
9137 141C      01630          DW      1C14H,141DH,1D15H,151CH
     1D14 151D 1C15
913F 2B        01640 RESET    DEC     HL
9140 7E        01650          LD      A,(HL)
9141 3C        01660          INC     A
9142 28B4      01670          JR      Z,MOVE
9144 18F9      01680          JR      RESET
               01690 ;
9146 23        01700 LASER    INC     HL              ;SETUP ALIEN BULLET
9147 C5        01710          PUSH    BC
9148 FD21E292  01720          LD      IY,BTAB
914C 0605      01730          LD      B,5             ;5 MAXIMUM
914E FD7E00    01740 BLP      LD      A,(IY)
9151 B7        01750          OR      A
9152 200E      01760          JR      NZ,AGAIN
9154 7A        01770          LD      A,D
9155 3C        01780          INC     A
9156 FD7701    01790          LD      (IY+1),A
9159 FD7302    01800          LD      (IY+2),E
915C FD360001  01810          LD      (IY),1
9160 1808      01820          JR      AGAIN1
9162 FD23      01830 AGAIN    INC     IY
9164 FD23      01840          INC     IY
9166 FD23      01850          INC     IY
9168 10E4      01860          DJNZ    BLP
916A C1        01870 AGAIN1   POP     BC
916B 7E        01880          LD      A,(HL)
916C C9        01890          RET
               01900 ;
916D DD21C492  01910 PALIEN   LD      IX,ALIEN        ;DISPLAY ALIEN
9171 0605      01920          LD      B,5
9173 DD7E00    01930 PLP      LD      A,(IX)
9176 B7        01940          OR      A
9177 2814      01950          JR      Z,FLIP
9179 DD5601    01960          LD      D,(IX+1)
917C DD5E02    01970          LD      E,(IX+2)
917F CDAE92    01980          CALL    GETL
9182 DD7E00    01990          LD      A,(IX)
9185 3D        02000          DEC     A
9186 280D      02010          JR      Z,A1
9188 3D        02020          DEC     A
9189 2818      02030          JR      Z,A2
918B 1824      02040          JR      A3
918D 110500    02050 FLIP     LD      DE,5
9190 DD19      02060          ADD     IX,DE
9192 10DF      02070          DJNZ    PLP
9194 C9        02080          RET
               02090 ;GRAPHIC BLOCKS MAKE ALIEN SHAPE
9195 FD36009D  02100 A1       LD      (IY),157        ;*
9199 FD3601B3  02110          LD      (IY+1),179      ;*
919D FD3602AE  02120          LD      (IY+2),174      ;*
91A1 18EA      02130          JR      FLIP
91A3 FD3600AD  02140 A2       LD      (IY),173        ;*
91A7 FD36018F  02150          LD      (IY+1),143      ;*
91AB FD36029E  02160          LD      (IY+2),158      ;*
91AF 18DC      02170          JR      FLIP
91B1 FD3600A6  02180 A3       LD      (IY),166        ;*
91B5 FD3601BF  02190          LD      (IY+1),191      ;*
```

```
91B9 FD360299 02200          LD      (IY+2),153      ;*
91BD 18CE     02210          JR      FLIP
              02220 ;
91BF DD21C492 02230 VALIEN   LD      IX,ALIEN        ;VERIFY ALIEN
91C3 0605     02240          LD      B,5
91C5 DD7E00   02250 VLP      LD      A,(IX)
91C8 B7       02260          OR      A
91C9 2819     02270          JR      Z,PASS
91CB DD5601   02280          LD      D,(IX+1)
91CE DD5E02   02290          LD      E,(IX+2)
91D1 CDAE92   02300          CALL    GETL
91D4 AF       02310          XOR     A
91D5 FDBE00   02320          CP      (IY)
91D8 2812     02330          JR      Z,EX
91DA FDBE01   02340          CP      (IY+1)
91DD 280D     02350          JR      Z,EX
91DF FDBE02   02360          CP      (IY+2)
91E2 2808     02370          JR      Z,EX
91E4 110500   02380 PASS     LD      DE,5
91E7 DD19     02390          ADD     IX,DE
91E9 10DA     02400          DJNZ    VLP
91EB C9       02410          RET
91EC FD3600BF 02420 EX       LD      (IY),WHITE
91F0 FD3601BF 02430          LD      (IY+1),WHITE
91F4 FD3602BF 02440          LD      (IY+2),WHITE
91F8 DD360000 02450          LD      (IX),0
91FC 3AF592   02460          LD      A,(ALEFT)
91FF 3D       02470          DEC     A
9200 32F592   02480          LD      (ALEFT),A
9203 18DF     02490          JR      PASS
              02500 ;
9205 DD21E292 02510 BULLET   LD      IX,BTAB         ;MOVE BULLETS DOWN
9209 0605     02520          LD      B,5
920B DD7E00   02530 BLP2     LD      A,(IX)
920E B7       02540          OR      A
920F 2816     02550          JR      Z,TRY
9211 DD5601   02560          LD      D,(IX+1)
9214 DD5E02   02570          LD      E,(IX+2)
9217 14       02580          INC     D
9218 7A       02590          LD      A,D
9219 FE11     02600          CP      17              ;*BULLET LIMIT
921B 2813     02610          JR      Z,BOFF
921D DD7201   02620          LD      (IX+1),D
9220 CDAE92   02630          CALL    GETL
9223 FD360088 02640          LD      (IY),136        ;*BULLET SHAPE
9227 DD23     02650 TRY      INC     IX
9229 DD23     02660          INC     IX
922B DD23     02670          INC     IX
922D 10DC     02680          DJNZ    BLP2
922F C9       02690          RET
9230 DD360000 02700 BOFF     LD      (IX),0
9234 18F1     02710          JR      TRY
              02720 ;
9236 3AF592   02730 CALIEN   LD      A,(ALEFT)       ;CHECK ALIEN
9239 B7       02740          OR      A
923A C0       02750          RET     NZ
923B 3E05     02760          LD      A,5
923D 32F592   02770          LD      (ALEFT),A
9240 0605     02780          LD      B,5
9242 DD21C492 02790          LD      IX,ALIEN
9246 1601     02800          LD      D,1
9248 1E05     02810          LD      E,5
924A 3AF492   02820          LD      A,(TYPE)
924D 3C       02830          INC     A
924E FE04     02840          CP      4
9250 CC7192   02850          CALL    Z,FIX
9253 32F492   02860          LD      (TYPE),A
9256 CD7792   02870          CALL    PAT
9259 3AF492   02880 CLP      LD      A,(TYPE)
925C DD7700   02890          LD      (IX),A
```

```
925F CD9992    02900          CALL    LOAD
9262 7E        02910          LD      A,E
9263 0E0B      02920          LD      C,11            ;*SPACE BETWEEN ALIENS
9265 81        02930          ADD     A,C
9266 5F        02940          LD      E,A
9267 D5        02950          PUSH    DE
9268 110500    02960          LD      DE,5
926B DD19      02970          ADD     IX,DE
926D D1        02980          POP     DE
926E 10E9      02990          DJNZ    CLP
9270 C9        03000          RET
9271 CD8992    03010 FIX      CALL    INIT
9274 3E01      03020          LD      A,1
9276 C9        03030          RET
9277 3D        03040 PAT      DEC     A
9278 2807      03050          JR      Z,P1
927A 3D        03060          DEC     A
927B 2808      03070          JR      Z,P2
927D 210493    03080          LD      HL,PAT3
9280 C9        03090          RET
9281 21F992    03100 P1       LD      HL,PAT1
9284 C9        03110          RET
9285 210293    03120 P2       LD      HL,PAT2
9288 C9        03130          RET
               03140 ;
               03150 ;SUBROUTINE PACKAGE:
9289 D9        03160 INIT     EXX                     ;CLEAR ALIEN BUFFER
928A 21C492    03170          LD      HL,ALIEN
928D 11C592    03180          LD      DE,ALIEN+1
9290 012C00    03190          LD      BC,44
9293 3600      03200          LD      (HL),0
9295 EDB0      03210          LDIR
9297 D9        03220          EXX
9298 C9        03230          RET
               03240 ;
9299 DD7503    03250 LOAD     LD      (IX+3),L        ;LOAD IX
929C DD7404    03260          LD      (IX+4),H
929F DD7201    03270          LD      (IX+1),D
92A2 DD7302    03280          LD      (IX+2),E
92A5 C9        03290          RET
               03300 ;
92A6 3A4038    03310 PAUSE    LD      A,(KEY)         ;WAIT ON "ENTER"
92A9 CB47      03320          BIT     0,A             ;*
92AB 28F9      03330          JR      Z,PAUSE
92AD C9        03340          RET
               03350 ;
92AE C5        03360 GETL     PUSH    BC              ;ARRAY ROUTINE
92AF 42        03370          LD      B,D
92B0 FD21C03B  03380          LD      IY,VDU-COLUMN
92B4 EB        03390          EX      DE,HL
92B5 114000    03400          LD      DE,COLUMN
92B8 FD19      03410 GLP      ADD     IY,DE
92BA 10FC      03420          DJNZ    GLP
92BC 4D        03430          LD      C,L
92BD 0600      03440          LD      B,0
92BF FD09      03450          ADD     IY,BC
92C1 EB        03460          EX      DE,HL
92C2 C1        03470          POP     BC
92C3 C9        03480          RET
               03490 ;VARIABLES:
001E           03500 ALIEN    DEFS    30              ;ALIEN BUFFER
000F           03510 BTAB     DEFS    15              ;BULLET BUFFER
92F1 0000      03520 LOC      DW      0               ;PLAYER'S LOCATION
92F3 00        03530 ME       DB      0               ;SHIPS LEFT
92F4 00        03540 TYPE     DB      0               ;TYPE OF ALIEN
92F5 00        03550 ALEFT    DB      0               ;ALIEN STILL ATTACKING
92F6 0000      03560 FLOC     DW      0               ;PLAYER'S LASER LOCATION
92F8 00        03570 COUNT    DB      0               ;PLAYER'S SHOT COUNTER
               03580 ;ALIEN ATTACK PATTERNS:
```

```
92F9 FF          03590 PAT1     DB       255,2,3,3,3,254,4,4,4
     02 03 03 03 FE 04 04 04
9302 FF          03600 PAT2     DB       255,2
     02
9304 FF          03610 PAT3     DB       255,5,5,5,5,6,6,6,254,6
     05 05 05 05 06 06 06 FE
     06
930E FF          03620          DB       255
9000             03630          END      START
00000 Total errors
```

I do not expect you to have grasped every important aspect of the program on the first reading (if you have, you are certainly above us mortals) but careful examination will bring to light many of its important features – and drawbacks. There is a lot of material to digest! Again, I emphasize that the program is only a simple one, much watered down from other pieces of source code to make it more comprehensible. There is an infinite number of ways to write such a game. This is only one.

5.5 GAME TECHNIQUE

This final section will tie up loose ends and expand upon some of the principles introduced in the earlier sections.

Throughout this chapter we have been working with byte units, although high and low resolution computers alike are capable of working with bits. Bits are used differently by each computer. Some are used to define the colour of a graphic block, the actual shape (they are then called pixels), and other definitions. The graphic blocks we have been working with so far, have six pixels: bits 0-5, with bit 6 reset and bit 7 set (if reset, it reverts to ASCII). Refer to diagram 5.4 again. This is a fairly simple design for a basic computer, with no need to define colour or other special attributes.

When a byte is displayed on the screen only the pixels appear. When one of these bits are set, a small block lights up to represent it. See diagram 5.6.

When we display whole bytes on the screen, and then move whole bytes across the screen, we skip (at least in this example) three pixels, moving up or down and two pixels, moving right or left. If you have the opportunity to play arcade games on many different computers, note the degrees of 'jumpyness' caused by this pixel skipping. Many programmers have overcome this problem by writing special routines to simulate pixel graphics. Real pixel graphics, setting bits instead of displaying bytes, is too messy and slow. Few programmers bother to do it.

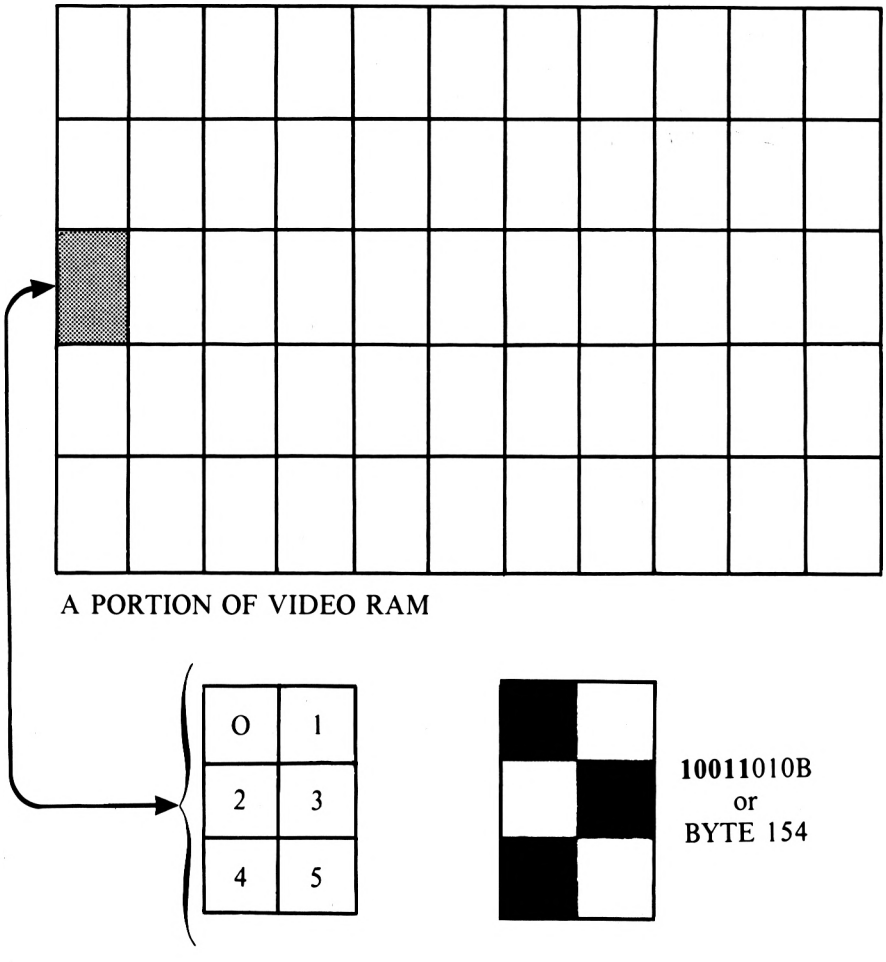


DIAGRAM 5.6 – VIDEO RAM PIXEL FORMAT

In some rare cases, the scale of bytes on the screen is equivalent to an average computer pixel. In which case, the programmer can program such super high resolution computers without worrying too much about individual pixels. For the majority of us being able to work with pixels is essential for high quality graphics.

I cannot solve your problem. I can show how to solve one problem. From this, hopefully, you will be able to write a routine for your own special needs. With this limitation in mind, we might as well look at a difficult problem.

There are 3 * 2 pixels in each byte of our 1K of VDU RAM, which makes for 6144 pixels all together; or a matrix of 48 pixel rows by 128 pixel columns. This is a substantial increase from our previous 64 * 16 byte matrix. The problem: A routine that will return the required address of a 48 * 128 pixel display, and the row and column of the pixel within that address. A modified GETL routine does it:

```
;CALCULATE BIT COORDINATES
;ENTRY -->    D = Y-AXIS (1-48)
;             E = X-AXIS (1-128)
;EXIT  -->    IY = ADDRESS OF ARRAY
;             A  = POINTS TO 1-3 PIXEL ROW

GETX    LD      HL,VDU-COLUMN
        LD      BC,COLUMN
        XOR     A
XLP     ADD     A,3             ;PIXEL WIDTH
        ADD     HL,BC
        CP      D               ;OVER LIMIT?
        JR      C,XLP
        SUB     D               ;GET A
        CPL                     ;INVERT BITS
        AND     3               ;MASK BITS 7-2
        LD      D,0             ;D FINISHED
        RES     0,E             ;MAKE EVEN
        SRA     E               ;DIVIDE BY 2
```

```
ADD     HL,DE
PUSH    HL
POP     IY
RET
```

Since 16 * 3 = 48, the instructions in XLP count sets of three instead of one. The accumulator is used as a counter, keeping track of the pixel column. When the correct column is reached, the D register is subtracted from A, leaving two, one or zero in the register, pointing to rows one, two or three in that order. What we really want is a number in A corresponding to the row width. To do this, we complement A and AND it with three. This is best explained in binary.

Follow this through slowly: The number 00000001B (1) complemented is 11111110B (254), and ANDed by '3' (00000111B) yields 00000010B or '2'. Likewise '2' yields a value of '1', and '0' a value of '3'. So the accumulator returns 1-3 according to its 1-3 pixel position.

The pixel length is twice that of the byte length (two pixels in each byte) so we need only divide by two, to point to the corresponding byte. We already know that by shifting bits right (the SRA instruction) we divide by two, but this only works for multiples of two. To overcome this, it resets bit 0 of E, which will make it even if odd, or else leave it unchanged. After which it is possible to divide by two and get a valid result. On exit, the accumulator holds the pixel width (1-3). Note that no calculation was performed to work out the pixel length. This is trivial, since the number is odd if one or even if two. A bit test will determine this automatically.

As you can see, pixel conversion is certainly no laughing matter, but remember, the example used was a complex one.

With the pixel width in the accumulator, we can write a routine to display three or even six variations of the same 'alien'. Examine diagram 5.7. The 'alien' originally appears on the first column, but is displayed over two columns when A returns with '2' or '3'. To do this efficiently, the display routine in chapter 3 comes in handy. The effect of this is a smooth transition from one column to another, yet requires extra work in not only designing further 'alien' shapes, but in checking for hits (the alien is stretched over two columns) from the player's spaceship.

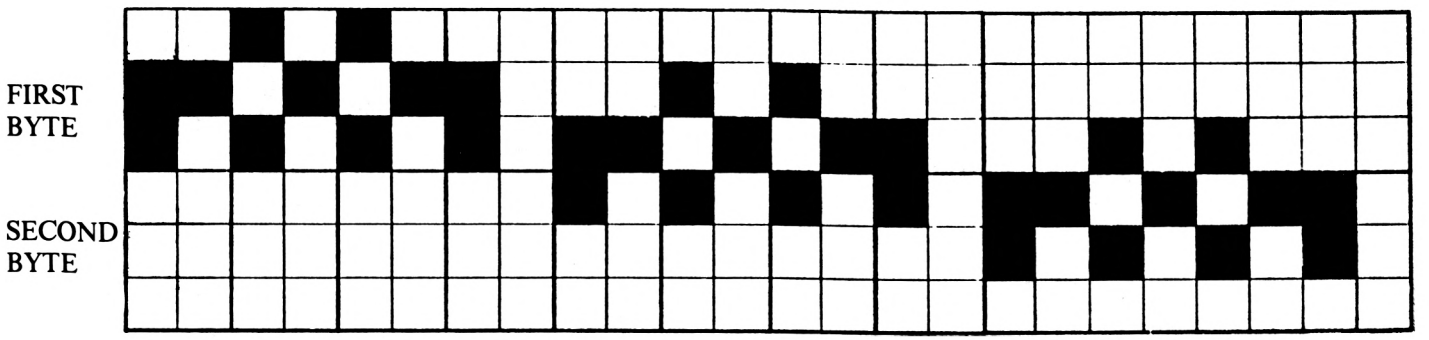


DIAGRAM 5.7 – SMOOTH GRAPHICS

Another problem becomes painfully obvious after a while. This is overlap. Anything close to the 'alien' will be blocked out by the graphic blocks placed on the column underneath.

The following routine displays only the pixels (by setting the matching bit) that are to be set on the screen. It must be modified to suit your own computer's pixel organisation. The principle, anyway, is what counts:

```
;DISPLAY PIXELS 0-5 IF SET
;ENTRY --->    A  = BYTE TO SET
;             HL = ADDRESS

BITSET LD     B,6          ;TRY 6 PIXELS
       LD     IX,TABLE     ;BIT CODES
BIT1   RRCA                ;PUT IN CARRY
       CALL   C,SET        ;SET IF ON
       INC    IX           ;NEXT
       DJNZ   BIT1
       RET
SET    PUSH   AF
       LD     A,(IX)       ;INSTRUCTION
       LD     (INST),A
       POP    AF
       DB     0CBH         ;SET X,(HL)
INST   NOP                 ;THE 'X'
       RET
```

```
;BITS 0,1,2,3,4,5:

TABLE   DB      0C6H,0CEH,0D6H

        DB      0DEH,0E6H,0EEH
```

Which brings to an end our discussion on graphics – and our incompatability problems. With these routines, or at least versions of them, you will be able to design computer games with graphics of the highest possible standard.

Arcade games often give scores that reach into the millions – well above the 65535 limit as demonstrated in Chapter 3. Short of writing a new routine, try adding a fake zero at the end of the displayed number. It is seldom that a player gets less than 10 points at a time, and it will bring the score limit up to 655350. You may be giving the player only five points, but it looks convincingly like 50.

A standard game incentive is to give away an 'extra' at 10000 or whatever points. The routine to do this is:

```
MORE    LD      HL,(SCORE)  ;CURRENT SCORE
        LD      DE,(EXTRA)  ;CURRENT EXTRA
        OR      A
        SBC     HL,DE       ;EQUAL?
        RET     C           ;RETURN CARRY
        LD      HL,10000    ;THE EXTRA
        ADD     HL,DE       ;NEW EXTRA
        LD      (EXTRA),HL
        .  .  .             ;INC SHIPS HERE
```

A carry indicates that the score has not yet exceeded the EXTRA. When it does, the EXTRA is adjusted to point to the next score needed to gain an extra spaceship.

The final routine in this chapter, refer to listing 5.3, sorts a list of 10 names according to score, in memory, for use in a high score table. The list can then be displayed on the VDU as 10 sets of a 16-bit number and 16 characters of ASCII.

The very last 16-bit number in the table should be compared to the latest one to determine if it ranks a place in the table. If it does, place the score and new name in the last data items and CALL SCORE.

Data is organised using a 'bubble sort', which will be discussed further in the next chapter.

```
                00100 ;SORT TOP TEN SCORES
                00110 ;VARIABLES:     B  = NO. OF NAMES-1
                00120 ;               DE = LENGTH OF NAMES+2
                00130 ;EXIT  -->      NAMES SORTED IN ASCENDING ORDER
                00140 ;*CHANGE ACCORDING TO COMPUTER
                00150 ;*****************************
9000            00160           ORG     9000H           ;*
                00170 ;
9000 DD215F90   00180 SCORE     LD      IX,HSC
9004 0609       00190           LD      B,9
9006 C5         00200 LOOP      PUSH    BC
9007 DDE5       00210           PUSH    IX              ;COPY IX INTO IY
9009 FDE1       00220           POP     IY
900B 111200     00230 LOOP2     LD      DE,18           ;LENGTH OF NAME+2
900E FD19       00240           ADD     IY,DE           ;POINT TO NEXT NAME
9010 DD6601     00250           LD      H,(IX+1)        ;GET SCORE
9013 DD6E00     00260           LD      L,(IX)
9016 FD5601     00270           LD      D,(IY+1)
9019 FD5E00     00280           LD      E,(IY)
901C B7         00290           OR      A
901D ED52       00300           SBC     HL,DE           ;COMPARE
901F 3033       00310           JR      NC,NOSWAP
9021 19         00320           ADD     HL,DE           ;SWAP SCORES
9022 FD7401     00330           LD      (IY+1),H
9025 FD7500     00340           LD      (IY),L
9028 DD7201     00350           LD      (IX+1),D
902B DD7300     00360           LD      (IX),E
902E DDE5       00370           PUSH    IX
9030 C5         00380           PUSH    BC
9031 FDE5       00390           PUSH    IY
9033 FD23       00400           INC     IY
9035 FD23       00410           INC     IY
9037 DD23       00420           INC     IX
9039 DD23       00430           INC     IX
903B 0610       00440           LD      B,16
903D DD5600     00450 FLIP      LD      D,(IX)          ;SWAP ASCII
9040 FD5E00     00460           LD      E,(IY)
9043 DD7300     00470           LD      (IX),E
9046 FD7200     00480           LD      (IY),D
9049 DD23       00490           INC     IX
904B FD23       00500           INC     IY
904D 10EE       00510           DJNZ    FLIP
904F FDE1       00520           POP     IY
9051 C1         00530           POP     BC
9052 DDE1       00540           POP     IX
9054 10B5       00550 NOSWAP    DJNZ    LOOP2           ;NEXT NAME
9056 C1         00560           POP     BC
9057 111200     00570           LD      DE,18           ;POINT TO NEXT
905A DD19       00580           ADD     IX,DE
905C 10AB       00590           DJNZ    LOOP            ;AND AGAIN
905E C9         00600           RET
                00610 ;THE DATA (IN SCRAMBLED SEQUENCE):
905F 2823       00620 HSC       DW      9000
9061 42         00630           DB      'BETA           '
     45 54 41 20 20 20 20 20
     20 20 20 20 20 20 20 20
9071 1027       00640           DW      10000
9073 41         00650           DB      'ALPHA          '
     4C 50 48 41 20 20 20 20
     20 20 20 20 20 20 20 20
```

```
9083 401F      00660          DW      8000
9085 54        00670          DB      'THETA          '
     48 45 54 41 20 20 20 20
     20 20 20 20 20 20 20 20
9095 581B      00680          DW      7000
9097 44        00690          DB      'DELTA          '
     45 4C 54 41 20 20 20 20
     20 20 20 20 20 20 20 20
90A7 7017      00700          DW      6000
90A9 45        00710          DB      'EPSILON        '
     50 53 49 4C 4F 4E 20 20
     20 20 20 20 20 20 20 20
90B9 8813      00720          DW      5000
90BB 47        00730          DB      'GAMMA          '
     41 4D 4D 41 20 20 20 20
     20 20 20 20 20 20 20 20
90CB B80B      00740          DW      3000
90CD 53        00750          DB      'SIGMA          '
     49 47 4D 41 20 20 20 20
     20 20 20 20 20 20 20 20
90DD A00F      00760          DW      4000
90DF 50        00770          DB      'PI             '
     49 20 20 20 20 20 20 20
     20 20 20 20 20 20 20 20
90EF E803      00780          DW      1000
90F1 5A        00790          DB      'ZETA           '
     45 54 41 20 20 20 20 20
     20 20 20 20 20 20 20 20
9101 D007      00800          DW      2000
9103 4F        00810          DB      'OMEGA          '
     4D 45 47 41 20 20 20 20
     20 20 20 20 20 20 20 20
9000           00820          END     SCORE
00000 Total errors
```

CHAPTER 6
– WRITING FOR BUSINESS

Time is money. Which, by way of simple maths, makes machine code the only language for business software. A business package today must be written in machine language to be worthy of serious consideration. A word processor or database program (whether this is inventory management, a mailing list or a personal filing system) not written in machine language is only a cruel joke. High level languages impose unacceptable limitations, whether this is speed or available I/O. Only in machine language does a programmer have total control over the computer, and the ability to guarantee a fast and compact product.

This chapter is about the three important aspects of business programming; storing, organising and retrieving information: so in many respects it is a direct continuation of chapter four. Storing involves such information structures as data blocks and linked lists. Organising looks at methods of sorting. And retrieving involves searching and indexing. Much has been written and said about business programming. This chapter moves quickly from simple sorting techniques to language design, but does not claim to be complete in any way. Being realistic, only an encyclopedia of computer programming could claim that. However, the routines demonstrated here have either been proven by time, or offer novel and efficient approaches to old problems.

6.1 SORTING

There are times when data has to be processed to suit some requirement, such as a list of items in alphabetical or numeric order. Thousands of pages of text are dedicated to the principles of sorting, so this section has been selective in offering what is regarded as the easiest method (to program and understand) and – one of – the fastest. Moreover, there are examples of 'no-sort' routines that do away with much of the work that limits sorting algorithms.

The 'bubble sort' is generally considered the easiest sorting algorithm in use, but, unfortunately, one of the slowest. Named so because the higher (or lower, depending on the routine) values 'bubble' to the top of the array, while the lower sink to the bottom.

A list of terms in *ascending* order start with the smallest value and progress, such as; 1, 5, 32, 111, 240. While a list of terms in *descending* order begin with the largest; 150, 81, 32, 5, and decrease. A sort, by reversing the compare instructions, can be made to organise information in either order.

The bubble sort is a sort of 'exchange', which means it swaps terms out of place in the list. There are other kinds of sorting, but exchanges offer a good speed and memory compromise. Bubble sorting is nevertheless slow. Doubling the number of terms to sort, squares the time needed to complete it. For short lists of 10 or 20 terms there is little problem, since the Z80 is fast enough to make even the most grossly inefficient routine seem fast. Yet a list of a hundred or a thousand terms or more may take minutes to sort. With its advantages and limitations in mind, a close examination of the algorithm is called for.

The actual format of the algorithm is not strictly agreed upon. I will show you two, and let you settle for the one you prefer. The first algorithm (for ascending order):

1. Compare the upper term with the lower term.
2. If the bottom term is smaller than the above term, swap around.
3. Move down one term and repeat steps 1 and 2 until there are no more terms to be compared.
4. Repeat steps 1-3 again until no swapping takes place (sort completed).

A card analogy helps to visualize this process. Preferably get a pack of playing cards and follow along. Firstly, spread out ten cards randomly in a vertical row. This is our list. Each card is a term. Place a finger on the top card and compare it to the one underneath. Does the lower card have a smaller number on it? If so, swap the card positions. Now move your finger to the second card and so on, comparing and swapping until you reach the end of the list. When this happens start again from the top. When you can move your finger through the list without swapping, you have sorted successfully. Actually do this, it will help you understand the following machine language:

```
;BUBBLE SORT VERSION 1

;SORT 10 TERMS BETWEEN 1-255


BUBBLE  LD      IX,DATA     ;TABLE OF TERMS
        LD      B,9         ;NO. OF TERMS-1
LOOP    LD      A,(IX)      ;UPPER TERM
        CP      (IX+1)      ;COMPARE TO LOW
        CALL    NC,SWAP     ;LOW IS BIGGER?
```

```
        INC     IX              ;NEXT
        DJNZ    LOOP            ;10 TIMES
        OR      A               ;DONE?
        JR      NZ,BUBBLE       ;NO
        RET
SWAP    RET     Z               ;NO SWAP NEEDED
        LD      C,(IX+1)        ;SWITCH,
        LD      (IX),C          ;'EM AROUND
        LD      (IX+1),A
        XOR     A               ;STILL SORTING
        RET

DATA    DB      4,22,181,201,34
        DB      1,100,91,17,110
```

The second algorithm (still for ascending order) is:

1. Compare the upper term to each lower term, swapping when the lower term is smaller than the above term.
2. The top card should now have the smallest value in the list. Repeat step 1, but start comparing at the next lower card in the list.
3. When the upper term has become the last term the sort is complete.

Follow along with the card analogy again. Take your ten cards and spread them out like before. Place your finger on the first card and look at the card below. If smaller, swap positions. Keep your finger on the top card. Now look at the next card (the third). Swap if necessary. Compare every card with the first one. This, by the time you have compared it with all the other terms, should now have the lowest value. Move your finger down one card (to the second card) and compare it with

the lower cards. Keep on swapping where you have to. Repeat the compare with the third card, etc. By the time your finger has moved to the last card, all will be sorted.

```
;BUBBLE SORT VERSION 2

;SORT 10 TERMS BETWEEN 1-255


BUBBLE LD     IX,DATA     ;TABLE OF TERMS
       LD     B,9         ;NO. OF TERMS-1
LOOP   PUSH   BC
       PUSH   IX          ;PUT IX IN IY
       POP    IY
LOOP1  INC    IY
       LD     A,(IX)      ;COMPARE ABOVE,
       CP     (IY)        ;TO LOWER
       CALL   NC,SWAP
       DJNZ   LOOP1       ;ALL TERMS
       INC    IX          ;NEXT
       POP    BC          ;RECOVER
       DJNZ   LOOP
       RET
SWAP   RET    Z           ;NOT IF EQUAL
       LD     C,(IY)      ;SWITCH 'EM
       LD     (IX),C
```

```
        LD      (IY),A
        RET

DATA    DB      45,111,12,1,200
        DB      58,255,18,111,3
```

So far the routines have dealt with organising numeric information. We can modify the latter routine to sort information in alphabetical order. The routine presumes that each string is 10 bytes in length.

```
;BUBBLE SORT VERSION 3
;SORT 10 BYTE LENGTH ASCII STRINGS

STR     EQU     10              ;STRING LENGTH
BUBBLE  LD      IX,DATA         ;TABLE OF TERMS
        LD      B,9             ;NO. OF TERMS-1
        LD      DE,STR          ;STRING LENGTH
LOOP    PUSH    BC              ;SAVE COUNTER
        PUSH    IX              ;PUT IX IN IY
        POP     IY
LOOP1   ADD     IY,DE           ;NEXT STRING
        PUSH    BC
        LD      B,STR           ;COMPARE
        PUSH    IX              ;SAVE POINTERS
        PUSH    IY
```

```
LOOP2   LD      A,(IX)          ;CP ASCII
        CP      (IY)
        JR      Z,NEXT          ;MAYBE?
        JR      C,PASS          ;NO WAY - SKIP
        POP     IY              ;GET START,
        POP     IX              ;OF STRINGS
        PUSH    IX
        PUSH    IY
        LD      B,STR           ;SWAP 'EM
SWAP    LD      C,(IY)
        LD      A,(IX)
        LD      (IX),C
        LD      (IY),A
        INC     IX
        INC     IY
        DJNZ    SWAP
        JR      PASS
NEXT    INC     IX              ;TRY NEXT
        INC     IY              ;CHARACTER
        DJNZ    LOOP2
PASS    POP     IY              ;NEXT TERM
        POP     IX
```

```
        POP     BC
        DJNZ    LOOP1           ;AGAIN
        POP     BC
        ADD     IX,DE           ;NEXT TERM
        DJNZ    LOOP
        RET
DATA    DB      'P. BOWER   ','J. HOWES   '
        DB      'J. NUGENT  ','D. ANGUS   '
        DB      'F. ALOE    ','J. MURPHY  '
        DB      'V. BLACK   ','C. CHOY    '
        DB      'E. MACLEAN','S. MAJOR   '
```

The routine is complicated by the need to compare the entire string, not just a single byte, and then relocate it. But it is essentially similar to the earlier routines, once you can follow the hierarchical stack operations. The compare in LOOP 2 checks both equal to and greater than, making sure that 'JOHN SMITH' and 'JOHN JONES' would swap positions, since the 'J' in Jones comes before the 'S' in Smith.

The 'Shell Sort', a kind of super-bubble sort, takes almost half as long as conventional bubble sorting. Using this method, we can double the number of items to sort with only slightly more than double the time. Which is a significant improvement over the four times longer required by a standard bubble sort. The algorithm however, while fine on paper, is not the most straightforward to implement.

The algorithm is:

1. Divide the list in two.
2. Compare the terms in the first list to the terms in the second list, swapping where appropriate.
3. Redivide the list, making the second list half its previous size.
4. Repeat step 3 until the second list is one term in length (list sorted).

The biggest problem is trying to divide the list into two sublists. What if the first list has an odd number of terms, say, 11? Or what happens when 10 terms are divided into two, five bytes sublists, and then redivided into half this? I opted for incrementing odd numbers, to make even, and then dividing. So that half of five yields three, or half of eleven yields six. Now for the card analogy. Spread out your ten cards again (in the vertical row) and place a finger on the top card and a finger on the sixth.

You have successfully divided your list into two sublists. Compare the first card to the sixth, swapping if the lower card is smaller, and then move on to the second and seventh, third and eighth, etc. Once your bottom finger has reached the last card, stop (but still make a final swap if necessary). Place your finger on the top card again and one on the third card. Why the third card? You have to divide your list into two sections again, but the new upper list must be only half the size of the previous one. Exchange the cards in a similar fashion again, comparing and swapping. Divide by two again, which leaves you with one (avoid confusion: the new subdivision is really two, but remember never to count the top card). Notice now that this has degenerated to a normal 'version 1' bubble sort. This final sort will finish off the sorting process.

The Shell sort is faster because really big values near the top get pushed down far more quickly. You would have noticed this while switching cards. In the prior sorts a large value was moved down only one term at a time, while in this sort greatly misplaced terms can traverse half the list in a single *pass*.

Experimenting with different sized lists, or even counting the compares in the sort demonstrated, makes a problem obvious. The comparing is not consistent. In one *pass* we might have to compare ten terms, in another five or six. The only way to determine if the list has been completely compared is to check that the computer is not looking beyond the last term in the list. We do this by subtracting. First the program:

```
;SHELL SORT

;SORT 10 TERMS BETWEEN 1-255

SHELL   LD      DE,5          ;HALF THE LIST
        LD      BC,DATA+10    ;END OF LIST
LOOP    LD      IX,DATA       ;START OF LIST
        LD      IY,DATA       ;   "         "
```

```
        ADD     IY,DE           ;SUBDIVIDE LIST
LOOP1   LD      A,(IX)          ;COMPARE
        CP      (IY)
        CALL    NC,SWAP
        INC     IX              ;NEXT TERM
        INC     IY
        PUSH    IY              ;CHECK FOR,
        POP     HL              ;LAST TERM
        OR      A               ;BY SUBTRACT
        SBC     HL,BC
        JR      NC,SKIP         ;LAST TERM?
        JR      LOOP1
SKIP    LD      A,E             ;ONLY 1 TERM,
        DEC     A               ;IN SUBLIST?
        RET     Z               ;DONE!
        BIT     0,E             ;IS E ODD?
        JR      Z,EVEN
        INC     E               ;MAKE EVEN
EVEN    SRA     E               ;DIVIDE BY 2
        JR      LOOP            ;DO IT AGAIN
SWAP    RET     Z               ;SWAP 'EM
        LD      L,(IY)
```

```
        LD      (IX),L
        LD      (IY),A
        RET

DATA    DB      45,12,111,1,200
        DB      156,2,12,100,87
```

Initially register E is loaded with the first subdivision of the list, exactly half the number of terms, which is added to IY. Register pair BC points to the end of the list. This is subtracted from HL, which holds the contents of IY. There will be an overflow if the end of the list has not yet been reached. Otherwise a JR NC terminates the compare loop and the E register is checked for one. If the sublist has degenerated to one term, as indicated by E, the routine RETs. The E register, if even, is divided by two, or made even (by incrementing it) and then divided. It now holds the new subdivision of the list. The routine repeats.

Before continuing, let's review a 'no-sort' method of organising information. Much of the time taken up by the bubble sort is in rearranging data, which is more obvious with strings than single bytes. Rather than move the string around in memory, as we did in the string bubble sort, we could assign it a number according to its order. By increasing the length of the string to 11 bytes, we could reserve the last byte as an indicator of the terms position in the list:

JAME	2
JOHN	3
ADAM	0
FRANK	1

This could be done by bubble sorting, incrementing the eleventh byte each time the string is compared to a lower one. This would leave the lowest with zero, having never been incremented, and the largest with the largest number. This is much faster to sort, but there is a problem in accessing the information. We would have to block compare 10 terms an average of 50 times.

This is reasonable for a small list. Alternately an index, pointing to the location of each string, could be derived from data in the eleventh byte. However, by the time this is done – it uses extra memory as well – it may be faster to bubble or Shell sort the list. (Note: If this method is implemented, we can carry out a very efficient binary search; more on this later.) At first glance the method does not seem

worthwhile, but it does have advantages that will become more apparent later on in the chapter.

The final sort, 'Quicksort', is one of the fastest, recommended for lists of a hundred or more terms. For smaller lists, say around ten, the standard Shell or even bubble sort is faster. The sort is the most sophisticated one we have looked at, so a lot of concentration is needed.

The algorithm:

1. Select an approximate median of the terms in the list, placing it at the top of the list.
2. Starting at the bottom of the list, move up, searching for a smaller term. Compare to the median: swap if necessary.
3. Once swapped, start at the top of the list moving down, comparing the current term with the previously swapped term; swap if necessary.
4. Continue to reverse the order of search. Stop when comparing the same card.
5. Divide the list into two sublists. One above the current term, one below the current term. The current term is in neither list.
6. Sort the two sublists as if they were two separate lists, starting from step 1 again.
7. When a sublist has no terms left, go to another list. When all sublists have no terms the sort is complete.

The card analogy will require some tricky finger work. Spread out your ten cards. First we must find the median. (The median is the 'middle term', so in the list; 2, 5, 9, the 5 is the median. In the list; 13, 22, 86, 95, 1000, the 86 is the median.) We can only determine the 'true' median by sorting the list and looking at the middle term – which is a bit pointless, since we want to Quicksort it, not bubble sort it. Therefore we have to approximate.

Look at the top, bottom, and middle card, and find the median of these. Swap this card with the top card (unless it *is* the top card). Remember this card. Now, place a finger on the top card and a finger on the bottom. Compare the top and bottom, and swap if necessary. If you did not swap the cards move your bottom finger up one card (second last) and compare again. Keep moving your finger up the list until you have to swap. If you did swap, move your top finger down one card (second card), keeping your other finger where it is! Now compare. If you did not need to swap, then keep moving it down. If you did swap, reverse the order of your search.

In summary, if you are moving your top finger down and swap a card, leave your finger there, but start searching from your bottom finger, moving it up. If you are moving your finger up and swap a card, leave your finger there, but start searching from your top finger, moving it down. Ultimately both fingers will meet (if it was a

fairly 'true' median, this will be somewhere in the middle) on the same card. If you have done it right, this card was the median you had previously selected. (If you didn't do it right, do it again. I found it a difficult algorithm to master as well.)

With the exception of luck, your cards will still lack numerical order, but the sort is not finished yet. Divide your list into two sublists (other books may call this *partitioning*), with one list above the card you have your fingers on, and one below. The card itself does not belong in either of the lists because it is already in its correct position. Looking at the above list you will notice that all its terms are less than (unless you swapped the other way around, in which case they will all be greater than) the approximate median. All the terms in the lower list will be greater than the median.

So we have successfully subdivided the list into 'greater thans', and 'smaller thans'. Treat the sublists as if they were two separate lists, sorting them in exactly the same way shown; even to the extent of selecting another median. What you get will largely depend in what cards you spread out and the median picked. You may get two sublists, three or even four or more. A sublist without any remaining terms is completely sorted. Go to another sublist and sort that. Once all sublists have been sorted in this manner, the entire list will be ordered. As you can see, for ten cards, this sort is complicated and lengthy. For a big sort there is only one word to describe it: *fast*.

```
;QUICKSORT

;SORT 10 BYTE-LENGTH TERMS


QUICK   LD      IX,DATA      ;1ST TERM
        LD      BC,9         ;NO. TERMS-1
        PUSH    IX           ;ACTUAL ROUTINE
        POP     IY           ;GET LAST TERM
        ADD     IY,BC        ;ADD TO IY
LOOP    LD      (FTERM),IX   ;1ST TERM
        LD      (LTERM),IY   ;LAST TERM
```

```
        LD      B,0
        CALL    SUBIX       ;NO TERMS LEFT?
        RET     Z           ;BACK
        RET     C           ;OR LESS THAN 1
        CALL    FMED        ;FIND MEDIAN
LOOP2   CALL    SUBIX       ;FINISHED?
        JR      Z,NEW       ;TERMS SAME?
        LD      A,(IX)      ;COMPARE TERMS
        CP      (IY)
        JR      C,PASS      ;NO SWAP
        JR      Z,PASS      ;NO SWAP
        LD      D,(IX)      ;SWAP 'EM
        LD      A,(IY)
        LD      (IX),A
        LD      (IY),D
        INC     B           ;SWITCH
PASS    BIT     0,B         ;EVEN OR ODD?
        JR      Z,LOOK      ;LOOK OTHER WAY
        INC     IX          ;DOWN 1 TERM
        INC     IY          ;CANCEL OUT
LOOK    DEC     IY          ;MOVE UP
        JR      LOOP2
```

```
NEW     PUSH   IY           ;SAVE OTHER,
        LD     IX,(LTERM)   ;LIST ON,
        PUSH   IX           ;STACK AND,
        DEC    IY           ;DO TOP ONES,
        LD     IX,(FTERM)   ;FIRST
        CALL   LOOP         ;SORT TOP LIST
        POP    IY           ;AND NOW THE,
        POP    IX           ;REST OF THEM
        INC    IX           ;SKIP MIDDLE,
        CALL   LOOP         ;AND DO AGAIN
        RET                 ;FINISHED ALL!
FMED    PUSH   BC           ;GET MEDIAN
        CALL   SUBIX        ;HL IS LENGTH
        PUSH   HL           ;PUT IN BC
        POP    BC
        LD     A,C          ;IF TERMS LESS,
        CP     4            ;THAN 4, NO,
        JR     C,BACK       ;MEDIAN NEEDED
        RES    0,C          ;MAKE EVEN
        SRA    C            ;DIVIDE BY 2
        PUSH   IX           ;PUT IN HL
        POP    HL           ;MIDDLE TERM
```

```
        ADD     HL,BC       ;IN HL
        POP     BC
AGAIN   LD      A,(IX)      ;COMPARE TERMS
        CP      (HL)
        JR      C,NEXT      ;FIRST TERM,
        CP      (IY)        ;MEDIAN?
        RET     Z
        JR      NC,WRONG    ;NOT IT
        RET
NEXT    CP      (IY)        ;MAYBE?
        RET     Z
        RET     NC
WRONG   LD      A,(HL)      ;SWITCH,
        LD      E,(IX)      ;TERMS,
        LD      D,(IY)      ;AROUND
        LD      (HL),D
        LD      (IX),A
        LD      (IY),E
        JR      AGAIN
BACK    POP     BC
        RET                 ;ABORT
SUBIX   PUSH    IY          ;SUBTRACT,
```

```
        PUSH    IX              ;IY FROM IX
        POP     DE
        POP     HL
        OR      A
        SBC     HL,DE
        RET


FTERM   DW      0
LTERM   DW      0
DATA    DB      13,201,44,8,91
        DB      200,50,100,1,1
```

The routine is exactly 169 bytes in length, a fair measure larger than all the others. On entry IX points to the start of the data table, with C holding the number of terms to be sorted minus one (to sort more than 255 terms B can be used, but another register must take its place in the routine LOOP1). The contents of IX are copied into IY and BC is added to it, which points it to the location of the last term in the list. A compare is made to determine whether the registers point to the same term or have crossed over each other; in which case the sort is aborted.

A median is then selected by calling FMED. The length of the list is calculated by subtracting IX from IY (although B already holds the length of the list, this will vary from sublist to sublist). The length is then divided by two to determine the middle term. If the list is even, say 10 terms, the term closest to zero is selected as the middle; in this case five. The routine AGAIN compares the first term to the second and third. To be the median, the first term must be greater than the second and smaller than the third, or, smaller than the second and greater than the third. If the first term is not the median, all three terms are swapped around and recompared.

Note, if there are three or less terms in the whole list, it makes little difference if a median is selected or not, so the routine exits automatically. The routine LOOP1 does all the swapping of terms and searching. When the same terms are being compared the routine jumps to NEW. This PUSHes the *other* sublist on the stack to

be processed later. The whole procedure repeats, PUSHing the locations of more and more sublists on the stack until one is completed – in which case the address of a previous sublist is POPed off and processed. When they have all been POPed off the routine RETs, with a sorted list. This type of coding is called *recursive* programming because subroutines repeatedly CALL themselves.

If you are wondering how much stack space is used, consider it enough to worry about. Four bytes are pushed on the stack for every sublist, so reserve a reasonable amount of RAM. The ideal sort would Quicksort the bulk of data until sublists are down to a size of 10 terms (put a compare in LOOP after CALL SUBIX), after which it can be Shell or bubble sorted in a few passes. That is the fastest possible way. Unless the list is massive, I would dismiss this need for the extra coding.

6.2 INFORMATION STRUCTURES

The last section dealt with organising information. This section discusses storing, finding, adding and deleting it. So far we have placed data in simple arrays of one or two dimensions, but this is not sufficient for more sophisticated data storage.

We sort information to make it easier to locate, but how can a computer take advantage of this structure as well? Humans put items in alphabetical order for good reasons. If we are looking for 'JOHN' and found 'JAMES' close by we would have an approximate idea of its location. Of course, we don't know how many J names are between the two records, but we can estimate this by knowing the length of the file. Likewise, a computer can be made to approximate the location of a record in a sorted file, once it knows the length.

An efficient method of locating the correct term in an ordered list is called *binary searching*. Thumbing through a dictionary we tend to disregard more and more pages as we narrow down the sought-after word. In an unordered dictionary, this approach would be impractical, to say the least. A binary search examines the middle term of the list, by dividing it in half, and checks which section it could not possibly be in, either because the next – or previous – term is too great or too small. It then disregards this half and redivides the remainder. As the search repeatedly subdivides the list, it narrows down to one possibility.

```
;BINARY SEARCH

;ENTRY -->    BC = SIZE OF LIST
```

```
;               HL = START OF LIST
;               D  = TERM TO BE LOCATED
;EXIT   --->    HL = ADDRESS OF TERM

START   LD      HL,DATA     ;LIST IN MEMORY
        LD      BC,10
        LD      D,5
        CALL    BFIND
        .   .   .
BFIND   SRL     B           ;DIVIDE BC BY 2
        RR      C
        ADD     HL,BC       ;GET MEDIAN
        SRL     B           ;DIVIDE AGAIN
        RR      C
        LD      A,(HL)      ;GET TERM
        CP      D
        RET     Z           ;FOUND IT?
        JR      NC,SUB      ;SMALLER?
        ADD     HL,BC       ;IT IS LARGER
        INC     HL          ;IN CASE BC = 0
        JR      BFIND
SUB     OR      A
```

```
        SBC     HL,BC           ;SMALLER
        DEC     HL              ;IN CASE BC = 0
        JR      BFIND


DATA    DB      1,2,3,4,5,6,7,8,9,10
```

The routine BFIND divides the list into two halves. Then redivides the remainder and keeps this as the amount to add or subtract next. A compare discards the 'greater than' or 'smaller than' part of the list, and the addition or subtraction is made. In case the contents of BC degenerates to zero before the term is found, an increment or decrement follows the subtraction or addition. The routine expects the term to be in the list.

You will probably need to store more than just a name or number in memory. A database program keeps its information in a 'record' or block of information. Each block would be divided into a number of 'fields' like name, address, and other pertinent data. In an inventory program we might want to locate a certain model car, with certain features, in a certain colour. Or we might want to make a list of all vehicles according to model, and one according to manufacturer. A simple and effective way of setting up such a file is to keep each record in a block of bytes, kept in some ordered sequence. If each block is of the same length we can continue to binary search it. For example:

```
BLOCK   DB      'Z80B          '
        DB      'ZILOG INC               '
        DB      '123-X-112C'
        DW      2710H
```

could be a typical record. First we have the name of the item, reserving 10 bytes, the name of the manufacturer, reserving 20 bytes, the catalog number to help us locate it, occupying 10 bytes, and the number in stock – two bytes non-ASCII. We could sort this list according to item, manufacturer, catalogue number or amount in stock by comparing fields and moving the rest of the information mandatorily with it (this is what happens in the Top Ten routine in chapter five). A particular ASCII string could be found using a block compare:

```
;BLOCK COMPARE
;ENTRY    --> HL = START OF LIST
;              BC = NUMBER OF TERMS
; ADDRESS (LEN) = LENGTH OF STRING
;              DE = LENGTH OF DATA BLOCK
;              IX = START OF STRING
;EXIT     --> HL = ADDRESS OF TERM
;      CARRY SET IF MATCHED
START  LD     HL,DATA     ;LIST IN MEMORY
       LD     A,12
       LD     (LEN),A
       LD     BC,3
       LD     DE,15
       CALL   FIND
       .  .  .

FIND   PUSH   BC
       PUSH   HL
       LD     A,(LEN)     ;LENGTH IN BC
       LD     C,A
       LD     B,Ø
LOOP   LD     IX,BUFFER   ;STRING
```

```
LOOP1   LD      A,(IX)          ;GET ASCII
        INC     IX              ;NEXT
        CPI                     ;BLOCK COMPARE
        JP      PO,GOTIT        ;STRING MATCHED
        JR      Z,LOOP1         ;YES, SO FAR...
        POP     HL              ;RECOVER START,
        POP     BC              ;AND COUNTER
        ADD     HL,DE           ;NEXT BLOCK
        DEC     BC              ;ONE LESS LOOP
        LD      A,B             ;FINISHED?
        OR      C
        JR      NZ,FIND
        RET                     ;YES, NO ENTRY
GOTIT   SCF                     ;SET CARRY
        POP     HL              ;GET START OF,
        POP     BC              ;BLOCK
        RET                     ;FOUND

LEN     DB      0
DATA    DB      'DEBBIE ANGUS'
        DB      36,24,33
        DB      'JULIE MURPHY'
```

```
        DB      34,23,36
        DB      'SHARON MAJOR'
        DB      35,25,36
BUFFER  DB      'JULIE MURPHY'
```

Or a binary search routine based on this data structure would also be suitable.

What is evident with the former routine (although not evident with the latter routine while still applicable), is that a data structure of a preset size wastes bytes. A name field must reserve a reasonable amount of memory, whether a name is 'SUE' or 'SUSANNA' or whether the item is a 'Z80B' or an 'EXPANSION INTERFACE'. Each field must reserve as many bytes as the largest entry, and resort to abbreviations if something larger appears. Changing the structure, examine the following:

```
BLOCK   DB      'Z80B',255
        DB      'ZILOG INC',255
        DB      '123-X-112C',255
        DW      2710H
```

Here the information occupies only as much space as required, with data separated by a 255 and the record ending with a two byte number. Searching through the record is a matter of placing carefully positioned block compares in sequence for three sets of 255, and then skipping two bytes past the 16-bits of data. Compares face more overhead to calculate the different record lengths, and so would be fractionally slower. But in terms of sheer memory saving, this would seem to be the only logical choice, even when records of a preset length are faster to search and easier to sort.

Yet there are other factors to be considered, not obvious when discussing machine code. How is this information going to be stored on a peripheral? Using a cassette recorder or a tape cartridge drive, both systems are equally suitable. But data has to be read 'sequentially' from the mass storage device. So to get the last record, we have to read the entire file. There is also however 'random access' where sectors of a disk drive are reserved for specific records.

We can therefore read a set of sectors without having to read the whole file. Both methods present problems in updating. To update a sequential file, we have to load

it all into memory (part by part) and rewrite the file. The 'random access' file cannot be updated, not at least without assigning an old record with a new record (an extension), but it is significantly faster.

These problems do not appear unless the file is larger than addressable memory and can only be accessed one piece at a time. If you need to store consumptious amounts of information; as in a mailing list or a database management aid, then I advise reserving a preset number of bytes for each record. This method is less complex to keep track of.

If your record keeping is limited to the extent of your computer's memory, such as line numbers in a computer language, divider and nil bytes are more flexible.

A block of information can be maintained as a simple array or as a linked-list (sometimes called a *chain*) where each record points to the location of the next one. Refer to diagram 6.0.

A record in a linked-list has a two byte address reserved as a pointer to the next record. An address of 0FFFFH or 0000H at the end of a record would be interpreted as meaning the last record in the file. A typical linked-list record would be:

```
BLOCK   DB      'Z80B',255
        DB      'ZILOG INC',255
        DB      '123-X-112C',255
        DW      2710H
        DW      4590H           ;ADDRESS
```

The data ends with a two byte address pointing to the next record. There is no need to relocate data which, theoretically, can be placed in a different sequence by readjusting pointers. We could insert a record by changing the pointers of the old record to point to the new record, then change the new record to point to the one the old record was previously addressing. To delete a record we make the previous pointers the same as the deleted records.

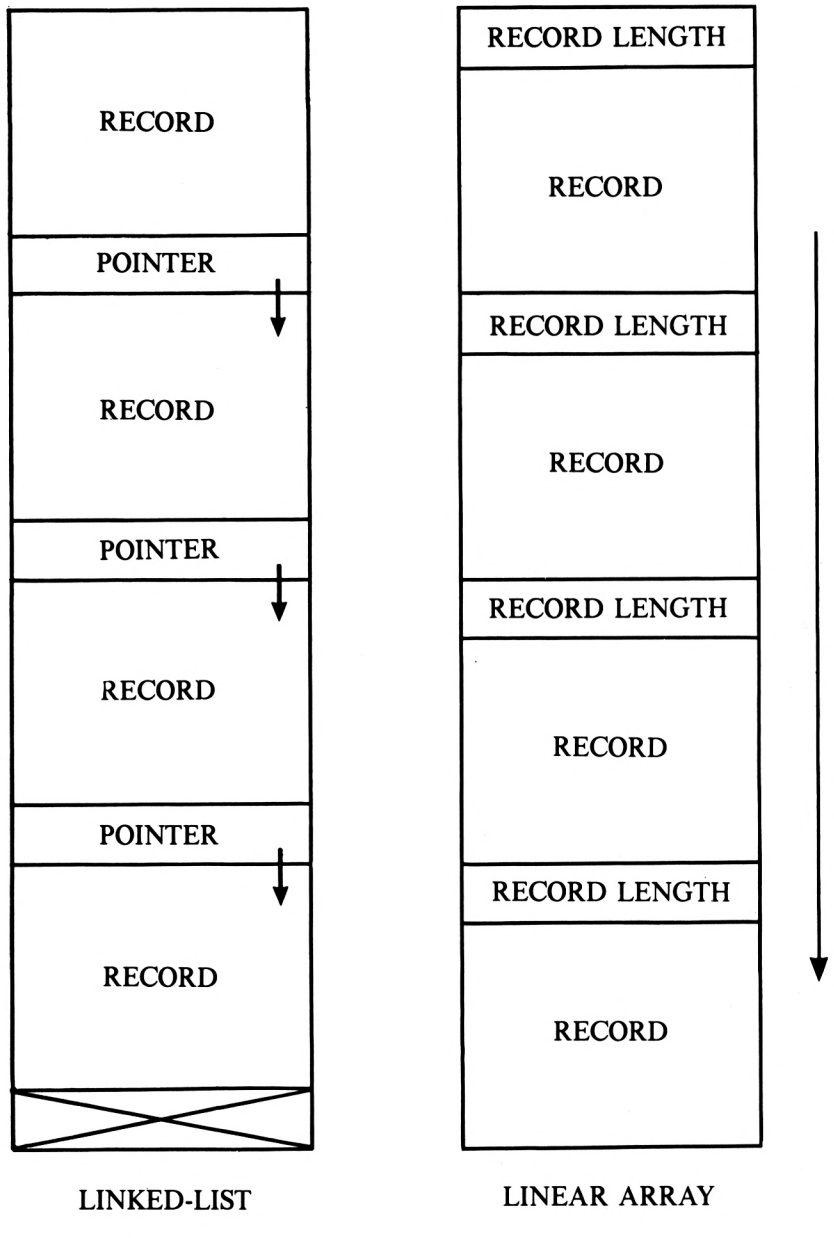


DIAGRAM 6.0 – DATA STRUCTURES

However, there must be some way of reusing the deleted record or the reserved memory goes to waste. This is where the linked-list structure begins to break down. If all the records were of a predefined size, we could place the deleted record on a 'free list'. When data is being added, the computer would refer to the 'free list' and use a previously deleted record, or, in the case of an empty 'free list', create a record from unused RAM (or connect all unused RAM to the 'free list' on initialization). When all the records are of different lengths, it no longer becomes practical.

If the record lengths differed, the new record would either not fit into the deleted one, or waste memory because it doesn't need all the bytes reserved for the old record. It would then be necessary to move a whole section of memory further up or down in RAM. Since the whole point of a linked-list is never needing to move data, the structure type has limitations.

A very simple linked-list structure would require an 'insert' and 'delete' routine. Pointer readjustment is fairly straightforward. The larger part of such a program would need to compare strings and locate the correct records.

The following routine, FIND, compares two strings, one in a buffer, the other in a record. It has two entry conditions. By clearing the accumulator, you can specify the first condition, which will locate the first record with a larger name field. This way the insert routine can insert the new record in the correct alphabetical order. Any other value in the accumulator will result in a straight byte-for-byte compare. The routine will then exit with the carry set if a match was found.

```
;ENTRY -->IF A = 0 COMPARE FOR INSERT
;          ELSE     COMPARE FOR DELETE
;EXIT   -->IF A = 0 ADDRESS OF RECORD
;                   MATCH IN IX
;                   OR CARRY SET
;          ELSE     ADDRESS OF PREVIOUS
;                   RECORD IN HL

FIND    LD     IX,HEADER   ;1ST RECORD
```

```
        LD      (SWITCH),A ;SAVE FOR LATER
LOOP    LD      HL,BUFFER  ;COMPARE FILE
        PUSH    IX         ;SAVE RECORD
        LD      A,(SWITCH) ;LOOKING FOR?
        OR      A          ;INSERT?
        JR      Z,FIND1    ;YES
        LD      B,15       ;15 BYTE ASCII
LOOP1   LD      A,(IX)     ;DO THE COMPARE
        CP      (HL)
        JR      NZ,WRONG   ;NO EXACT MATCH
        INC     HL         ;YES, TRY NEXT
        INC     IX
        DJNZ    LOOP1      ;AND AGAIN
        POP     IX         ;A MATCH!
        LD      DE,22      ;POINT TO,
        ADD     IX,DE      ;POINTERS
        SCF                ;SET CARRY
        RET                ;BACK
WRONG   POP     HL         ;GET LOCATION,
        LD      DE,22      ;OF NEXT,
        ADD     HL,DE      ;RECORD IN HL
        PUSH    HL
```

```
        LD      A,(HL)          ;GET ADDRESS
        INC     HL
        LD      L,(HL)
        XOR     L               ;IS IT ØFFFFH?
        POP     HL
        RET     Z               ;ABORT
        PUSH    HL              ;PUT ADDRESS,
        POP     IX              ;IN IX
        LD      (LAST),IX       ;SAVE PREVIOUS
        LD      H,(IX+1)        ;GET POINTERS
        LD      L,(IX)
        PUSH    HL
        POP     IX
        JR      LOOP            ;NEXT RECORD

;FIND ROUTINE FOR INSERT

FIND1   LD      B,16            ;LEN OF REC-1
LOOP2   DEC     B               ;DONE?
        JR      Z,WRONG
LOOP3   LD      A,(IX)          ;COMPARE
        CP      (HL)
        INC     HL              ;NEXT BYTES
```

```
        INC     IX
        JR      Z,LOOP3
        JR      C,WRONG
        POP     IX              ;MUST BE LARGER
        LD      HL,(LAST)       ;GET PREVIOUS
        RET                     ;DONE

LAST    DW      0
SWITCH  DB      0
MEM     DW      MEMTOP
BUFFER  DB      'TIM HARTNELL
        DB      255
        DB      1,2,3,4,5,6
HEADER  DC      21,0
        DB      255
        DW      B1
B1      DB      'ADAM MACREADIE '
        DB      255
        DB      2,3,4,6,7,1
        DW      B2
B2      EB      'BILL MACLEAN   '
        DB      255
```

```
        DB      9,8,1,3,1,0
        DW      B3
B3      DB      'TIM HARTNELL
        DB      255
        DB      1,8,5,2,1,2
        DW      0FFFFH
MEMTOP  DW      $
```

The FIND routine begins by loading IX with the *header* record. This is not a real record, nor can you insert before it or delete it (provided that only standard ASCII can be placed in the name field). This serves as a starting point for the compare routine.

The contents of the accumulator are saved away for later reference and HL is pointed to BUFFER, which has presumably been prepared by a data input routine (not shown here). Looking at the data blocks, B1, B2 and B3 you will see two fields, one of 15 bytes ASCII (the name field), a byte divider, and six bytes of other various numeric data. The compare routine examines the record according to the name field.

The LOOP routine compares the two strings. When a match is found, the start of the record is POPed off the stack and 22 bytes are added to it. This points it to the address that points to the next record.

In FIND1 a similar compare is made, but for a name larger than the one in the buffer. When this is found, HL is pointed to the previous record (the address is held in the address LAST) and it RETurns.

The routine WRONG finds the next record by reading the address at the end of the current record. It checks that this address is not 0FFFFH (or zero), which is the code for 'end of file', by loading A into IX and the routine repeats, else it aborts with carry not set.

Using this routine, it is possible to construct the 'insertion' and 'deletion' routines for the linked-list. To delete:

```
;DELETE RECORD IN A LINKED-LIST
```

```
DELETE  LD      A,1             ;FIND EXACT,
        CALL    FIND            ;MATCH
        RET     NC              ;NO MATCH
        LD      H,(IX+1)        ;CURRENT RECORD
        LD      L,(IX)
        LD      IX,(LAST)       ;PREVIOUS
        LD      (IX+1),H        ;NEXT RECORD
        LD      (IX),L
        RET
```

The routine RETurns if no match is found, otherwise the jump address pointing to the next record of the 'to be deleted' record is moved to the previous record, held in LAST. This record is therefore skipped everytime the file is read. If this routine is executed 'TIM HARTNELL' will be deleted from the file. Examine diagram 6.1.

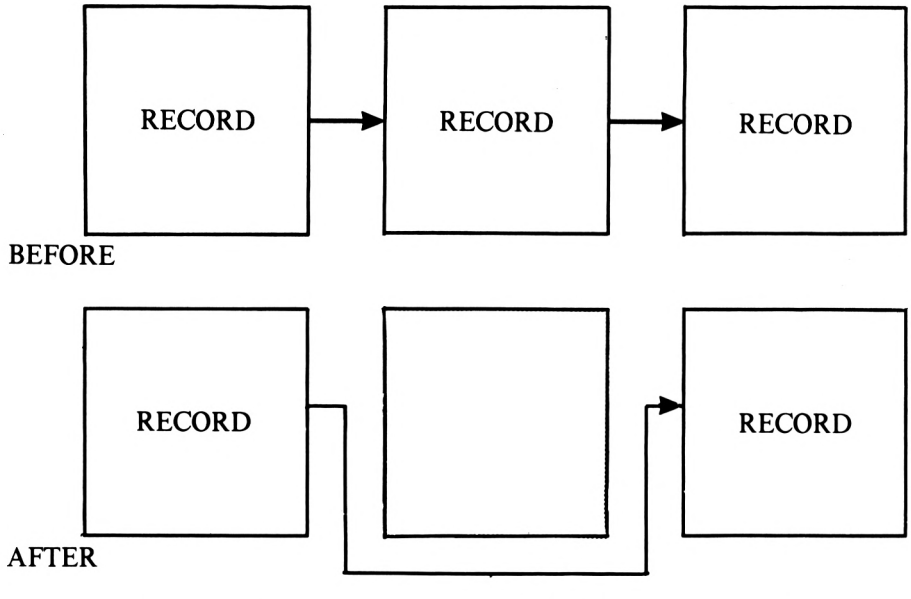


DIAGRAM 6.1 – DELETE A RECORD FROM A LINKED-LIST

In a more complex routine a 'free list' would have been implemented, connecting this address to other deleted ones for later use, rather than ignoring it and wasting 24 bytes.

The insert routine places the contents of BUFFER in alphabetical order in the linked-list (since it is impossible to sort a random length linked-list without moving data, this has to be a standard procedure). If there is nothing smaller in the list, it will not be inserted. You can modify this by changing the RET Z in the WRONG routine to jump to another section of code. In this section, load A with the contents of SWITCH, if non-zero RETurn, else load HL with the contents of LAST and then RETurn.

```
;INSERT RECORD IN A LINKED-LIST

INSERT XOR   A             ;FIND PREVIOUS,
       CALL  FIND          ;RECORD
       LD    DE,(MEM)      ;TOP OF MEMORY
       PUSH  HL
       POP   IY            ;PUT IN IY
       LD    (IY+1),D      ;ADJUST,
       LD    (IY),E        ;PREVIOUS
       LD    HL,BUFFER     ;GET NEW RECORD
       LD    BC,22         ;LENGTH
       LDIR                ;MOVE IT
       PUSH  IX            ;SWAP
       PUSH  DE
       POP   IX
       POP   DE
       LD    (IX+1),D      ;ADJUST POINTER
```

```
INC     IX              ;SKIP RECORD
INC     IX
LD      (MEM),IX        ;NEW MEM TOP
RET
```

The routine adjusts the old record to point to the new one, and the new one to point to what the previous one was pointing to. A new record is block moved into the top of memory (all memory above the routine) and the top of memory is moved past the new record, ready for the next insert. Examine diagram 6.2.

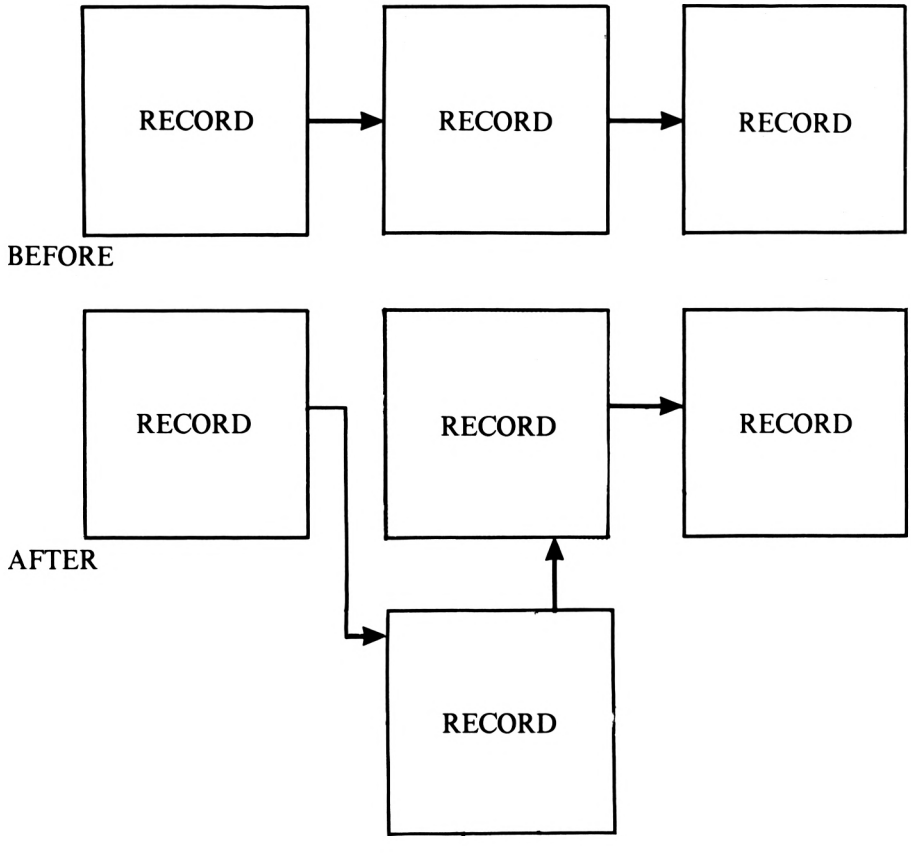


DIAGRAM 6.2 – INSERT A RECORD IN A LINKED-LIST

As you can see, even simple linked-lists require a lot of 'house keeping' to function. Although searching will slow down as the file grows, records can be added or deleted without a very noticeable delay.

A file of 20 or 30K organised as a simple linear array would delay for one or two seconds on a slow operating Z80 (say, around 2 MHz), since it must block move the entire file the length of the new record up or down in RAM. Many applications, from database management to new versions of popular programming languages, move data in this manner – even if the program has a 'type of' linked-list structure. After typing a large amount of source code into an editor/assembler or a computer language, you may possibly notice this slowing down of record processing.

6.3 COMPUTER LANGUAGES

This chapter finishes with the largest program in the book, listing 6.0. It has been specifically designed to maintain line numbers, including inserting, listing and deleting, for a computer language. What it lacks is an *interpreter.*

An interpreter decodes a language's high level instructions into something comprehensible – Z80 machine code. One is not included in this listing, because a readily convertible one has already been listed in chapter four: the English word decoder. Type in, convert to your machine, and execute the program. Typing 'L' (the instruction for *list*), followed by '0', will display the following:

```
10 AN EXAMPLE LINE

20 ANOTHER EXAMPLE

30 AND YET ANOTHER ONE
```

```
                00100 ;LINE NUMBER MAINTENANCE FOR A COMPUTER LANGUAGE
                00110 ;*PROGRAM USES STANDARD ASCII
                00120 ;*CHANGE ACCORDING TO COMPUTER
                00130 ;*****************************
                00140 ;
3C00            00150 VDU     EQU     15360         ;*START OF VDU RAM
0400            00160 LEN     EQU     1024          ;*LENGTH OF VDU RAM
3FC0            00170 BTL     EQU     16320         ;*BOTTOM LEFT-HAND CORNER
0040            00180 COLUMN  EQU     64            ;*COLUMN WIDTH
3880            00190 SHIFT   EQU     3880H         ;*ADDRESS OF SHIFT KEY
9000            00200         ORG     9000H         ;*
                00210 ;
9000 F3         00220 BEGIN   DI
9001 CD3D93     00230         CALL    INIT          ;SET UP REGISTERS & RAM
                00240 ;
9004 31FFFF     00250 RUN     LD      SP,65535      ;*EXECUTION CONTROL LOOP
9007 CDFE91     00260         CALL    CBUFF         ;CLEAR BUFFER
```

```
900A 3E3E     00270          LD       A,62             ;ASCII ">"
900C CD0E92   00280          CALL     SHOW             ;DISPLAY ON VDU
900F CD4190   00290          CALL     GETK             ;SCAN KEYBOARD
9012 CD7690   00300          CALL     WORK             ;CONVERT ASCII TO NO.
              00310 ;GET INSTRUCTION AND EXECUTE:
9015 211C94   00320          LD       HL,BUFFER        ;BUFFER = ALL KEY INPUT
9018 7E       00330          LD       A,(HL)           ;1ST BYTE 1 LETTER INST.
9019 010400   00340          LD       BC,4             ;NO. OF INST.+1
901C 21A293   00350          LD       HL,CTABLE        ;HOLDS ASCII INST.
901F 11FFFF   00360          LD       DE,0FFFFH        ;TO BE INCREMENTED
9022 EDA1     00370 FKEY     CPI                       ;BLOCK COMPARE
9024 13       00380          INC      DE               ;CALCULATE OFFSET
9025 E23990   00390          JP       PO,NOGOOD        ;NO COMPARES: BAD COMMAND
9028 20F8     00400          JR       NZ,FKEY          ;NOT IT, TRY NEXT COMMAND
902A 21A593   00410          LD       HL,VECTOR        ;JUMP TABLE OF COMMANDS
902D CB23     00420          SLA      E                ;MULTIPLY x 2
902F 19       00430          ADD      HL,DE            ;ADD OFFSET TO HL
9030 7E       00440          LD       A,(HL)           ;NOW GET (HL)
9031 23       00450          INC      HL
9032 66       00460          LD       H,(HL)
9033 6F       00470          LD       L,A
9034 110490   00480          LD       DE,RUN           ;SAVE RETURN ADDRESS
9037 D5       00490          PUSH     DE               ;ON STACK
9038 E9       00500          JP       (HL)             ;JUMP TO COMMAND
              00510 ;BAD COMMANDS HERE:
9039 21EC93   00520 NOGOOD   LD       HL,ERR3          ;"INVALID COMMAND"
903C CDB792   00530          CALL     PRINT            ;DISPLAY IT
903F 18C3     00540          JR       RUN              ;AND TRY AGAIN
              00550 ;GET INPUT AND PUT IN "BUFFER":
9041 211C94   00560 GETK     LD       HL,BUFFER
9044 06FA     00570          LD       B,250            ;LINE LENGTH IS 250
9046 CDD692   00580 GLP      CALL     GKEY             ;GET ASCII IN A
9049 FE08     00590          CP       8                ;ASCII BKSP
904B 2811     00600          JR       Z,BKSP           ;JUMP BACKSPACE
904D FE0D     00610          CP       13               ;ASCII CR
904F CA2692   00620          JP       Z,SCROLL         ;JUMP CARRIAGE RETURN
9052 1003     00630          DJNZ     GLP1             ;IF NOT FULL, GET A BYTE
9054 04       00640          INC      B                ;YES, FULL
9055 18EF     00650          JR       GLP              ;DON'T PUT IN BUFFER
9057 77       00660 GLP1     LD       (HL),A           ;PUT IN BUFFER
9058 CD0E92   00670          CALL     SHOW             ;DISPLAY AS WELL
905B 23       00680          INC      HL               ;NEXT PLACE
905C 18E8     00690          JR       GLP              ;AND AGAIN
905E 78       00700 BKSP     LD       A,B              ;CHECK FOR 250
905F FEFA     00710          CP       250
9061 28E3     00720          JR       Z,GLP            ;SKIP IF BUFFER EMPTY,
9063 2B       00730          DEC      HL               ;OTHERWISE, BACK UP
9064 3600     00740          LD       (HL),0           ;CLEAR IT
9066 04       00750          INC      B                ;BACK UP COUNTER
9067 E5       00760          PUSH     HL               ;FIX CURSOR,
9068 2A1794   00770          LD       HL,(LOC)         ;BY GETTING LOCATION,
906B 3620     00780          LD       (HL),32          ;AND CLEARING IT
906D 2B       00790          DEC      HL               ;BACK 1
906E 221794   00800          LD       (LOC),HL         ;SAVE IT
9071 365F     00810          LD       (HL),95          ;CURSOR GOES 1 BACK
9073 E1       00820          POP      HL               ;RECOVER
9074 18D0     00830          JR       GLP              ;AND DO IT AGAIN
              00840 ;THIS DECODES NUMBERS:
9076 211C94   00850 WORK     LD       HL,BUFFER
9079 0E00     00860          LD       C,0
907B 23       00870          INC      HL
907C 7E       00880          LD       A,(HL)
907D B7       00890          OR       A
907E 28B9     00900          JR       Z,NOGOOD         ;NO NUMBERS?
9080 7E       00910 SKIP     LD       A,(HL)           ;FIND FIRST NO.
9081 FE2C     00920          CP       ','              ;IS IT A COMMA?
9083 2814     00930          JR       Z,GOT1           ;YES! MUST BE 2 NUMBERS
9085 B7       00940          OR       A                ;IS IT ZERO?
9086 2809     00950          JR       Z,GOT2           ;MUST BE ONLY 1 NUMBER,
9088 23       00960          INC      HL               ;OTHERWISE KEEP LOOKING
9089 0C       00970          INC      C                ;NUMBER OF DIGITS IN C
```

```
908A 79         00980          LD      A,C
908B FE06       00990          CP      6               ;NOTHING OVER 6!
908D 2827       01000          JR      Z,BAD           ;NO. TO BIG FOR 16-BITS
908F 18EF       01010          JR      SKIP            ;KEEP LOOKING
9091 CD8192     01020 GOT2     CALL    ASCII           ;CONVERT ASCII TO A NUMBER
9094 FD221394   01030          LD      (LINE),IY       ;SAVE IN "LINE"
9098 C9         01040          RET
9099 E5         01050 GOT1     PUSH    HL              ;SAVE FOR 2ND NUMBER
909A CD9190     01060          CALL    GOT2            ;GET 1ST NUMBER
909D E1         01070          POP     HL              ;RECOVER
909E 23         01080          INC     HL              ;SKIP ","
909F 0E00       01090          LD      C,0             ;ZERO COUNTER
90A1 7E         01100 SKIP1    LD      A,(HL)          ;GET NO. OF DIGITS
90A2 B7         01110          OR      A               ;ZERO MEANS END
90A3 2809       01120          JR      Z,GOT3          ;END?
90A5 0C         01130          INC     C               ;KEEP LOOKING
90A6 79         01140          LD      A,C
90A7 FE06       01150          CP      6               ;NOTHING OVER 6!
90A9 280B       01160          JR      Z,BAD
90AB 23         01170          INC     HL
90AC 18F3       01180          JR      SKIP1
90AE CD8192     01190 GOT3     CALL    ASCII           ;CONVERT
90B1 FD221594   01200          LD      (LINE1),IY      ;SAVE IN "LINE1"
90B5 C9         01210          RET
90B6 21D793     01220 BAD      LD      HL,ERR2         ;"NUMBER OVERFLOW"
90B9 CDB792     01230          CALL    PRINT
90BC C30490     01240          JP      RUN
                01250 ;THIS "LISTS" LINE NUMBERS:
90BF DD211C95   01260 LIST     LD      IX,START        ;FIRST LINE NUMBER
90C3 CDDE91     01270 PLP1     CALL    LIMIT           ;NO LINES IN MEMORY?
90C6 D0         01280          RET     NC
90C7 DD5600     01290          LD      D,(IX)          ;PUT LINE NO. IN DE
90CA DD5E01     01300          LD      E,(IX+1)        ;16-BIT NUMBER
90CD 2A1394     01310          LD      HL,(LINE)       ;GET OPERATOR LINE
90D0 B7         01320          OR      A
90D1 ED52       01330          SBC     HL,DE           ;REACHED IT YET?
90D3 2802       01340          JR      Z,PLP3          ;YES
90D5 3021       01350          JR      NC,PASS         ;NO, SKIP THIS LINE
90D7 EB         01360 PLP3     EX      DE,HL           ;HL = LINE NUMBER
90D8 CD4C92     01370          CALL    DECX            ;DISPLAY NO. ON VDU
90DB 3E20       01380          LD      A,32            ;MOVE 1 SPACE ACROSS
90DD CD0E92     01390          CALL    SHOW            ;DISPLAY IT
90E0 DD4602     01400          LD      B,(IX+2)        ;LENGTH OF LINE
90E3 DD23       01410          INC     IX              ;SKIP 1ST BYTE OF LINE NO.
90E5 DD23       01420          INC     IX              ;SKIP 2ND
90E7 DD23       01430          INC     IX              ;SKIP LINE LENGTH
90E9 DD7E00     01440 PLP      LD      A,(IX)          ;ACTUAL DATA STARTS HERE
90EC CD0E92     01450          CALL    SHOW            ;DISPLAY IT
90EF DD23       01460          INC     IX              ;GO TO NEXT
90F1 10F6       01470          DJNZ    PLP             ;AND AGAIN
90F3 CD2692     01480          CALL    SCROLL          ;MOVE DISPLAY UP 1 ROW
90F6 18CB       01490          JR      PLP1            ;AGAIN
90F8 DD4E02     01500 PASS     LD      C,(IX+2)        ;SKIP THIS LINE
90FB 0600       01510          LD      B,0
90FD 03         01520          INC     BC
90FE 03         01530          INC     BC
90FF 03         01540          INC     BC
9100 DD09       01550          ADD     IX,BC           ;PASSED
9102 18BF       01560          JR      PLP1            ;TRY NEXT
                01570 ;THIS "DELETES" LINE NUMBERS:
9104 DD211C95   01580 DELETE   LD      IX,START        ;1ST LINE
9108 2A1394     01590 DLP      LD      HL,(LINE)       ;GET OPERATOR LINE NO.
910B DD5600     01600          LD      D,(IX)          ;LOAD DE WITH LINE NO.
910E DD5E01     01610          LD      E,(IX+1)
9111 B7         01620          OR      A
9112 ED52       01630          SBC     HL,DE           ;COMPARE
9114 3811       01640          JR      C,E1            ;NO SUCH LINE?
9116 2815       01650          JR      Z,FOUND         ;GOT IT!
9118 DD4E02     01660          LD      C,(IX+2)        ;TRY NEXT
911B 0600       01670          LD      B,0
```

```
911D 03       01680          INC     BC              ;SKIP 3 DATA BYTES,
911E 03       01690          INC     BC
911F 03       01700          INC     BC
9120 DD09     01710          ADD     IX,BC           ;AND ASCII
9122 CDDE91   01720          CALL    LIMIT           ;END OF LINES YET?
9125 38E1     01730          JR      C,DLP           ;NOT YET
9127 21C993   01740 E1       LD      HL,ERR1         ;"LINE DOESN'T EXIST"
912A C3B792   01750          JP      PRINT
912D DDE5     01760 FOUND    PUSH    IX              ;NOW DELETE IT...
912F E1       01770          POP     HL              ;PUT LINE IN HL
9130 DD4E02   01780          LD      C,(IX+2)        ;GET LENGTH,
9133 0600     01790          LD      B,0
9135 03       01800          INC     BC              ;AND ADD 3
9136 03       01810          INC     BC
9137 03       01820          INC     BC
9138 E5       01830          PUSH    HL              ;SHIFT REGISTERS,
9139 2A1194   01840          LD      HL,(TCODE)      ;AROUND FOR A BLOCK,
913C B7       01850          OR      A               ;MOVE
913D ED42     01860          SBC     HL,BC
913F 221994   01870          LD      (T1),HL         ;SAVE THIS ONE
9142 E1       01880          POP     HL
9143 09       01890          ADD     HL,BC
9144 E5       01900          PUSH    HL
9145 EB       01910          EX      DE,HL
9146 2A1194   01920          LD      HL,(TCODE)
9149 B7       01930          OR      A
914A ED52     01940          SBC     HL,DE
914C E5       01950          PUSH    HL
914D C1       01960          POP     BC
914E 03       01970          INC     BC
914F E1       01980          POP     HL
9150 DDE5     01990          PUSH    IX
9152 D1       02000          POP     DE
9153 CDC392   02010          CALL    MOVE            ;ALL READY, MOVE IT!
9156 2A1994   02020          LD      HL,(T1)         ;NEW END OF LINE NO.
9159 221194   02030          LD      (TCODE),HL      ;POINTER IN "TCODE"
915C C9       02040          RET
              02050 ;THIS "INSERTS" A LINE NUMBER:
915D DD211C95 02060 INSERT   LD      IX,START        ;START OF LINE NUMBER
9161 CDDE91   02070          CALL    LIMIT           ;ANY LINES IN MEMORY?
9164 301F     02080          JR      NC,NOLINE       ;NO, NOTHING TO BLOCK MOVE
9166 2A1394   02090 ILP      LD      HL,(LINE)       ;GET REQUESTED LINE
9169 DD5600   02100          LD      D,(IX)          ;PUT OTHER IN DE
916C DD5E01   02110          LD      E,(IX+1)
916F B7       02120          OR      A
9170 ED52     02130          SBC     HL,DE           ;COMPARE
9172 28B3     02140          JR      Z,E1            ;LINE ALREADY EXISTS!
9174 3840     02150          JR      C,DOIT          ;RIGHT SPOT TO INSERT
9176 DD4E02   02160          LD      C,(IX+2)        ;GET LENGTH
9179 0600     02170          LD      B,0
917B 03       02180          INC     BC              ;SKIP THOSE 3 BYTES
917C 03       02190          INC     BC
917D 03       02200          INC     BC
917E DD09     02210          ADD     IX,BC           ;AND THAT ASCII
9180 CDDE91   02220          CALL    LIMIT           ;OVER THE LIMIT?
9183 38E1     02230          JR      C,ILP           ;NOT YET
9185 CDE991   02240 NOLINE   CALL    GLINE           ;GET LINE OF DATA
9188 3A1B94   02250          LD      A,(LLEN)        ;LENGTH IN "LLEN"
918B C603     02260          ADD     A,3             ;SKIP 3
918D 4F       02270          LD      C,A
918E 0600     02280          LD      B,0
9190 2A1194   02290 BLOCK    LD      HL,(TCODE)      ;MOVE TO RIGHT SPOT
9193 09       02300          ADD     HL,BC           ;INC END POINTER
9194 221194   02310          LD      (TCODE),HL      ;SAVE IT
9197 211C94   02320          LD      HL,BUFFER       ;DATA TO MOVE
919A DDE5     02330          PUSH    IX
919C D1       02340          POP     DE              ;DESTINATION
919D 13       02350          INC     DE              ;THESE 3 BYTES,
919E 13       02360          INC     DE              ;TO BE FILLED WITH OTHER,
919F 13       02370          INC     DE              ;INFORMATION
```

```
91A0 3A1B94   02380          LD      A,(LLEN)        ;GET NEW LINES LENGTH
91A3 4F       02390          LD      C,A             ;PUT IN BC
91A4 EDB0     02400          LDIR                    ;MOVE IT IN
91A6 2A1394   02410          LD      HL,(LINE)       ;GIVE IT A LINE NUMBER
91A9 DD7400   02420          LD      (IX),H
91AC DD7501   02430          LD      (IX+1),L
91AF 3A1B94   02440          LD      A,(LLEN)        ;GIVE IT A LENGTH
91B2 DD7702   02450          LD      (IX+2),A
91B5 C9       02460          RET
91B6 CDE991   02470 DOIT     CALL    GLINE           ;GET LINE OF DATA
91B9 0600     02480          LD      B,0
91BB 3A1B94   02490          LD      A,(LLEN)        ;"LLEN" IS ITS LENGTH
91BE 4F       02500          LD      C,A
91BF 03       02510          INC     BC              ;SKIP 3 BYTES
91C0 03       02520          INC     BC
91C1 03       02530          INC     BC
91C2 C5       02540          PUSH    BC              ;SWAP REGISTERS,
91C3 DDE5     02550          PUSH    IX              ;AROUND,
91C5 E1       02560          POP     HL              ;TO BLOCK MOVE LINES,
91C6 EB       02570          EX      DE,HL           ;DOWN IN MEMORY
91C7 D5       02580          PUSH    DE
91C8 E1       02590          POP     HL
91C9 09       02600          ADD     HL,BC
91CA EB       02610          EX      DE,HL
91CB E5       02620          PUSH    HL
91CC D5       02630          PUSH    DE
91CD 2A1194   02640          LD      HL,(TCODE)
91D0 B7       02650          OR      A
91D1 ED52     02660          SBC     HL,DE
91D3 09       02670          ADD     HL,BC
91D4 E5       02680          PUSH    HL
91D5 C1       02690          POP     BC
91D6 D1       02700          POP     DE
91D7 E1       02710          POP     HL
91D8 CDC392   02720          CALL    MOVE            ;BLOCK MOVE 'EM
91DB C1       02730          POP     BC
91DC 18B2     02740          JR      BLOCK           ;NOW HAVE SPACE TO INSERT
              02750 ;
              02760 ;SUBROUTINE PACKAGE:
91DE DDE5     02770 LIMIT    PUSH    IX              ;THIS ROUTINE CHECKS THAT,
91E0 E1       02780          POP     HL              ;THE CURRENT LINE NO.,
91E1 ED5B1194 02790          LD      DE,(TCODE)      ;IS NOT OUT OF RANGE,
91E5 B7       02800          OR      A               ;(IS VALID)
91E6 ED52     02810          SBC     HL,DE           ;MUST HAVE AN OVERFLOW
91E8 C9       02820          RET
              02830 ;
91E9 2A1394   02840 GLINE    LD      HL,(LINE)       ;THIS GETS A LINE OF DATA
91EC CD4C92   02850          CALL    DECX            ;PRINT LINE NO. ON VDU,
91EF 3E20     02860          LD      A,32            ;AND A SPACE
91F1 CD0E92   02870          CALL    SHOW
91F4 CD4190   02880          CALL    GETK            ;NOW, TYPE DATA IN BUFFER
91F7 3EFA     02890          LD      A,250           ;CALCULATE LENGTH OF DATA,
91F9 90       02900          SUB     B
91FA 321B94   02910          LD      (LLEN),A        ;AND PUT IN "LLEN"
91FD C9       02920          RET
              02930 ;
91FE D9       02940 CBUFF    EXX                     ;THIS CLEARS THE BUFFER
91FF 211C94   02950          LD      HL,BUFFER
9202 111D94   02960          LD      DE,BUFFER+1
9205 01FB00   02970          LD      BC,251
9208 3600     02980          LD      (HL),0
920A EDB0     02990          LDIR
920C D9       03000          EXX
920D C9       03010          RET
              03020 ;
920E E5       03030 SHOW     PUSH    HL              ;THIS DISPLAYS,
920F D5       03040          PUSH    DE              ;A CHARACTER ON THE VDU
9210 2A1794   03050          LD      HL,(LOC)        ;"LOC" POINTS TO CURSOR
9213 77       03060          LD      (HL),A
9214 23       03070          INC     HL
```

```
9215 365F      03080          LD      (HL),95          ;LOAD CURSOR
9217 221794    03090          LD      (LOC),HL
921A 11FF3F    03100          LD      DE,VDU+LEN-1     ;END OF SCREEN?
921D B7        03110          OR      A
921E ED52      03120          SBC     HL,DE            ;COMPARE
9220 CC2692    03130          CALL    Z,SCROLL         ;SCROLL UP IF SO
9223 D1        03140          POP     DE
9224 E1        03150          POP     HL
9225 C9        03160          RET
               03170 ;
9226 D9        03180 SCROLL   EXX                      ;THIS SCROLLS THE VDU
9227 2A1794    03190          LD      HL,(LOC)
922A 3620      03200          LD      (HL),32
922C 21403C    03210          LD      HL,VDU+COLUMN    ;MOVE EVERYTHING UP
922F 11003C    03220          LD      DE,VDU
9232 0140FC    03230          LD      BC,VDU-BTL
9235 EDB0      03240          LDIR
9237 21C03F    03250          LD      HL,BTL           ;CLEAR BOTTOM LINE
923A 11C13F    03260          LD      DE,BTL+1
923D 013F00    03270          LD      BC,COLUMN-1
9240 3620      03280          LD      (HL),32
9242 EDB0      03290          LDIR
9244 21C03F    03300          LD      HL,BTL           ;PUT CURSOR BACK IN BTL
9247 221794    03310          LD      (LOC),HL
924A D9        03320          EXX
924B C9        03330          RET
               03340 ;
924C E5        03350 DECX     PUSH    HL               ;DISPLAYS A 16-BIT NO.
924D C5        03360          PUSH    BC               ;ON THE VDU
924E D5        03370          PUSH    DE
924F CD5692    03380          CALL    DEC
9252 D1        03390          POP     DE
9253 C1        03400          POP     BC
9254 E1        03410          POP     HL
9255 C9        03420          RET
9256 010005    03430 DEC      LD      BC,500H
9259 FD210794  03440          LD      IY,DTAB
925D 3E30      03450 DEC1     LD      A,30H
925F FD5601    03460          LD      D,(IY+1)
9262 FD5E00    03470          LD      E,(IY)
9265 B7        03480          OR      A
9266 ED52      03490 DEC2     SBC     HL,DE
9268 3803      03500          JR      C,DEC3
926A 3C        03510          INC     A
926B 18F9      03520          JR      DEC2
926D 19        03530 DEC3     ADD     HL,DE
926E FD23      03540          INC     IY
9270 FD23      03550          INC     IY
9272 FE30      03560          CP      30H
9274 2003      03570          JR      NZ,JUMP
9276 B9        03580          CP      C
9277 2005      03590          JR      NZ,OMIT
9279 CD0E92    03600 JUMP     CALL    SHOW
927C 0E30      03610          LD      C,30H
927E 10DD      03620 OMIT     DJNZ    DEC1
9280 C9        03630          RET
               03640 ;
9281 DDE5      03650 ASCII    PUSH    IX               ;CONVERT ASCII INTO A NO.
9283 DD210000  03660          LD      IX,0
9287 FD21FD93  03670          LD      IY,NTAB
928B 2B        03680 LP5      DEC     HL
928C 7E        03690          LD      A,(HL)
928D FE30      03700          CP      30H
928F DAB690    03710          JP      C,BAD
9292 FE3A      03720          CP      3AH
9294 D2B690    03730          JP      NC,BAD
9297 D630      03740          SUB     30H
9299 280E      03750          JR      Z,LP2
929B 47        03760          LD      B,A
929C FD5601    03770          LD      D,(IY+1)
```

```
929F FD5E00    03780          LD      E,(IY)
92A2 DD19      03790 LP3      ADD     IX,DE
92A4 DAB690    03800          JP      C,BAD
92A7 10F9      03810          DJNZ    LP3
92A9 FD23      03820 LP2      INC     IY
92AB FD23      03830          INC     IY
92AD 0D        03840          DEC     C
92AE 20DB      03850          JR      NZ,LP5
92B0 DDE5      03860          PUSH    IX
92B2 FDE1      03870          POP     IY
92B4 DDE1      03880          POP     IX
92B6 C9        03890          RET
               03900 ;
92B7 7E        03910 PRINT    LD      A,(HL)          ;DISPLAY A MESSAGE ON VDU
92B8 FE2E      03920          CP      '.'             ;CHECK FOR STOP BYTE
92BA CA2692    03930          JP      Z,SCROLL        ;SCROLL IF FOUND,
92BD CD0E92    03940          CALL    SHOW            ;OTHERWISE DISPLAY
92C0 23        03950          INC     HL
92C1 18F4      03960          JR      PRINT
               03970 ;
92C3 B7        03980 MOVE     OR      A               ;AUTOMATICALLY CALCULATES,
92C4 ED52      03990          SBC     HL,DE           ;CURRENT DIRECTION OF,
92C6 3804      04000          JR      C,BACK          ;BLOCK MOVE
92C8 19        04010          ADD     HL,DE
92C9 EDB0      04020          LDIR
92CB C9        04030          RET
92CC 19        04040 BACK     ADD     HL,DE
92CD 0B        04050          DEC     BC
92CE 09        04060          ADD     HL,BC
92CF EB        04070          EX      DE,HL
92D0 09        04080          ADD     HL,BC
92D1 EB        04090          EX      DE,HL
92D2 03        04100          INC     BC
92D3 EDB8      04110          LDDR
92D5 C9        04120          RET
               04130 ;COMPLETE MEMORY-MAPPED KEYBOARD DECODING ROUTINE:
92D6 D9        04140 GKEY     EXX
92D7 DDE5      04150          PUSH    IX
92D9 FDE5      04160          PUSH    IY
92DB CDEE92    04170          CALL    KBDEC           ;SCAN KEYBOARD
92DE 01401F    04180          LD      BC,8000         ;*DEBOUNCE DELAY,
92E1 F5        04190          PUSH    AF              ;FOR 2MHz
92E2 0B        04200 DELAY    DEC     BC
92E3 78        04210          LD      A,B
92E4 B1        04220          OR      C
92E5 20FB      04230          JR      NZ,DELAY
92E7 F1        04240          POP     AF
92E8 FDE1      04250          POP     IY
92EA DDE1      04260          POP     IX
92EC D9        04270          EXX
92ED C9        04280          RET
92EE DD219493  04290 KBDEC    LD      IX,KTAB         ;POINTS TO BYTES TO SCAN
92F2 1E07      04300          LD      E,7             ;*NO. OF BYTES TO CHECK
92F4 01FF01    04310          LD      BC,1FFH
92F7 D5        04320 KLP      PUSH    DE
92F8 DD5601    04330          LD      D,(IX+1)        ;PUT BYTE IN DE
92FB DD5E00    04340          LD      E,(IX)
92FE 1A        04350          LD      A,(DE)
92FF DD23      04360          INC     IX              ;POINT TO NEXT BYTE
9301 DD23      04370          INC     IX
9303 C5        04380          PUSH    BC
9304 B7        04390          OR      A
9305 C43093    04400          CALL    NZ,PRESS        ;*FIND WHICH KEY IS PRESSED
9308 C1        04410          POP     BC
9309 F5        04420          PUSH    AF
930A 78        04430          LD      A,B             ;NEXT ROW OF ASCII
930B C608      04440          ADD     A,8
930D 47        04450          LD      B,A
930E F1        04460          POP     AF
930F D1        04470          POP     DE
```

```
9310 B7       04480          OR      A              ;ANY KEY PRESSED?
9311 2006     04490          JR      NZ,OVER        ;YES
9313 1D       04500          DEC     E              ;SCAN NEXT ROW
9314 20E1     04510          JR      NZ,KLP         ;IF NON-ZERO
9316 B7       04520          OR      A              ;ANY KEY PRESSED?
9317 28D5     04530          JR      Z,KBDEC        ;TRY AGAIN
9319 47       04540 OVER     LD      B,A            ;NOW CHECK <SHIFT>
931A 3A8038   04550          LD      A,(SHIFT)
931D B7       04560          OR      A
931E 2002     04570          JR      NZ,FIX         ;FIX UP IF SHIFTED,·
9320 78       04580          LD      A,B            ;OTHERWISE RETURN WITH
9321 C9       04590          RET                    ;ASCII IN A
9322 78       04600 FIX      LD      A,B            ;WHICH KEYS TO ADJUST?
9323 FE3C     04610          CP      60             ;DON'T WORRY ABOUT,
9325 D0       04620          RET     NC             ;LOWERCASE
9326 FE30     04630          CP      48             ;BUT WORRY ABOUT NUMBER
9328 3803     04640          JR      C,FIX1         ;ROW
932A D610     04650          SUB     16             ;ADJUST BY 16
932C C9       04660          RET
932D C610     04670 FIX1     ADD     A,16           ;ADJUST BY 16
932F C9       04680          RET                    ;FOR STANDARD ASCII SET
9330 0C       04690 PRESS    INC     C              ;CHECK BITS IN BYTE
92E7 F1       04240          POP     AF
92E8 FDE1     04250          POP     IY
92EA DDE1     04260          POP     IX
92EC D9       04270          EXX
92ED C9       04280          RET
92EE DD219493 04290 KBDEC    LD      IX,KTAB        ;POINTS TO BYTES TO SCAN
92F2 1E07     04300          LD      E,7            ;*NO. OF BYTES TO CHECK
92F4 01FF01   04310          LD      BC,1FFH
92F7 D5       04320 KLP      PUSH    DE
92F8 DD5601   04330          LD      D,(IX+1)       ;PUT BYTE IN DE
92FB DD5E00   04340          LD      E,(IX)
92FE 1A       04350          LD      A,(DE)
92FF DD23     04360          INC     IX             ;POINT TO NEXT BYTE
9301 DD23     04370          INC     IX
9303 C5       04380          PUSH    BC
9304 B7       04390          OR      A
9305 C43093   04400          CALL    NZ,PRESS       ;*FIND WHICH KEY IS PRESSED
9308 C1       04410          POP     BC
9309 F5       04420          PUSH    AF
930A 78       04430          LD      A,B            ;NEXT ROW OF ASCII
930B C608     04440          ADD     A,8
930D 47       04450          LD      B,A
930E F1       04460          POP     AF
930F D1       04470          POP     DE
9310 B7       04480          OR      A              ;ANY KEY PRESSED?
9311 2006     04490          JR      NZ,OVER        ;YES
9313 1D       04500          DEC     E              ;SCAN NEXT ROW
9314 20E1     04510          JR      NZ,KLP         ;IF NON-ZERO
9316 B7       04520          OR      A              ;ANY KEY PRESSED?
9317 28D5     04530          JR      Z,KBDEC        ;TRY AGAIN
9319 47       04540 OVER     LD      B,A            ;NOW CHECK <SHIFT>
931A 3A8038   04550          LD      A,(SHIFT)
931D B7       04560          OR      A
931E 2002     04570          JR      NZ,FIX         ;FIX UP IF SHIFTED,
9320 78       04580          LD      A,B            ;OTHERWISE RETURN WITH
9321 C9       04590          RET                    ;ASCII IN A
9322 78       04600 FIX      LD      A,B            ;WHICH KEYS TO ADJUST?
9323 FE3C     04610          CP      60             ;DON'T WORRY ABOUT,
9325 D0       04620          RET     NC             ;LOWERCASE
9326 FE30     04630          CP      48             ;BUT WORRY ABOUT NUMBER
9328 3803     04640          JR      C,FIX1         ;ROW
932A D610     04650          SUB     16             ;ADJUST BY 16
932C C9       04660          RET
932D C610     04670 FIX1     ADD     A,16           ;ADJUST BY 16
932F C9       04680          RET                    ;FOR STANDARD ASCII SET
9330 0C       04690 PRESS    INC     C              ;CHECK BITS IN BYTE
92E7 F1       04240          POP     AF
92E8 FDE1     04250          POP     IY
```

```
92EA DDE1       04260          POP     IX
92EC D9         04270          EXX
92ED C9         04280          RET
92EE DD219493   04290 KBDEC    LD      IX,KTAB          ;POINTS TO BYTES TO SCAN
92F2 1E07       04300          LD      E,7              ;#NO. OF BYTES TO CHECK
92F4 01FF01     04310          LD      BC,1FFH
92F7 D5         04320 KLP      PUSH    DE
92F8 DD5601     04330          LD      D,(IX+1)         ;PUT BYTE IN DE
92FB DD5E00     04340          LD      E,(IX)
92FE 1A         04350          LD      A,(DE)
92FF DD23       04360          INC     IX               ;POINT TO NEXT BYTE
9301 DD23       04370          INC     IX
9303 C5         04380          PUSH    BC
9304 B7         04390          OR      A
9305 C43093     04400          CALL    NZ,PRESS         ;#FIND WHICH KEY IS PRESSED
9308 C1         04410          POP     BC
9309 F5         04420          PUSH    AF
930A 78         04430          LD      A,B              ;NEXT ROW OF ASCII
930B C608       04440          ADD     A,8
930D 47         04450          LD      B,A
930E F1         04460          POP     AF
930F D1         04470          POP     DE
9310 B7         04480          OR      A                ;ANY KEY PRESSED?
9311 2006       04490          JR      NZ,OVER          ;YES
9313 1D         04500          DEC     E                ;SCAN NEXT ROW
9314 20E1       04510          JR      NZ,KLP           ;IF NON-ZERO
9316 B7         04520          OR      A                ;ANY KEY PRESSED?
9317 28D5       04530          JR      Z,KBDEC          ;TRY AGAIN
9319 47         04540 OVER     LD      B,A              ;NOW CHECK <SHIFT>
931A 3A8038     04550          LD      A,(SHIFT)
931D B7         04560          OR      A
931E 2002       04570          JR      NZ,FIX           ;FIX UP IF SHIFTED,
9320 78         04580          LD      A,B              ;OTHERWISE RETURN WITH
9321 C9         04590          RET                      ;ASCII IN A
9322 78         04600 FIX      LD      A,B              ;WHICH KEYS TO ADJUST?
9323 FE3C       04610          CP      60               ;DON'T WORRY ABOUT,
9325 D8         04620          RET     NC               ;LOWERCASE
9326 FE30       04630          CP      48               ;BUT WORRY ABOUT NUMBER
9328 3803       04640          JR      C,FIX1           ;ROW
932A D610       04650          SUB     16               ;ADJUST BY 16
932C C9         04660          RET
932D C610       04670 FIX1     ADD     A,16             ;ADJUST BY 16
932F C9         04680          RET                      ;FOR STANDARD ASCII SET
9330 0C         04690 PRESS    INC     C                ;CHECK BITS IN BYTE
93AB 43         05060 M1       DB      'COMPUTER LANGUAGE LINE NUMBERS'
     4F 4D 50 55 54 45 52 20
     4C 41 4E 47 55 41 47 45
     20 4C 49 4E 45 20 4E 55
     4D 42 45 52 53
93C9 49         05070 ERR1     DB      'INVALID LINE!.'
     4E 56 41 4C 49 44 20 4C
     49 4E 45 21 2E
93D7 4E         05080 ERR2     DB      'NUMBER OUT OF RANGE!.'
     55 4D 42 45 52 20 4F 55
     54 20 4F 46 20 52 41 4E
     47 45 21 2E
93EC 49         05090 ERR3     DB      'INVALID COMMAND!.'
     4E 56 41 4C 49 44 20 43
     4F 4D 4D 41 4E 44 21 2E
                05100 ;
                05110 ;USED BY DECIMAL CONVERSION ROUTINES:
93FD 0100       05120 NTAB     DW      1,10,100,1000,10000
     0A00 6400 E803 1027
9407 1027       05130 DTAB     DW      10000,1000,100,10,1
     E803 6400 0A00 0100
                05140 ;
                05150 ;VARIABLES:
9411 5695       05160 TCODE    DW      MEM              ;LAST LOCATION OF LINE NO.
9413 0000       05170 LINE     DW      0                ;STORAGE FOR USER INST.
9415 0000       05180 LINE1    DW      0                ;ADDITIONAL STORAGE
```

```
9417 0000      05190 LOC      DW      0              ;LOCATION OF CURSOR
9419 0000      05200 T1       DW      0              ;TEMPORARY STORAGE
941B 00        05210 LLEN     DB      0              ;TEMPORARY LINE LENGTH
941C 00        05220 BUFFER   DB      0              ;BUFFER STARTS HERE
               05230 ;
951C           05240          ORG     $+255          ;ENDS HERE
               05250 ;EXAMPLE LINE FORMAT:
               05260 ;2 BYTES LINE NO. 1 BYTE LINE LENGTH. REST ASCII
951C 00        05270 START    DB      0,10,15,'AN EXAMPLE LINE'
     0A 0F 41 4E 20 45 58 41
     4D 50 4C 45 20 4C 49 4E
     45
952E 00        05280          DB      0,20,15,'ANOTHER EXAMPLE'
     14 0F 41 4E 4F 54 48 45
     52 20 45 58 41 4D 50 4C
     45
9540 00        05290          DB      0,30,19,'AND YET ANOTHER ONE'
     1E 13 41 4E 44 20 59 45
     54 20 41 4E 4F 54 48 45
     52 20 4F 4E 45
9556 5595      05300 MEM      DW      $-1            ;ADDRESS FOR "TCODE"
9000           05310          END     9000H
00000 Total errors
```

You can experiment by deleting these lines; type 'D' followed by the line number, or inserting; type 'I' followed by a line number. Any number between zero and 65535 is permissable. Higher or negative or unreadable (letters mixed with numbers) will display an error message and the instruction will be ignored. Nor will it allow you to insert over a line that already exists. Other instructions are treated as invalid commands.

Examine a possible line number:

```
10 CLS
```

In memory this would look like:

```
DB      0,10          ;LINE NUMBER

DB      3             ;LINE LENGTH

DB      'CLS'         ;3 BYTES ASCII
```

The first two bytes hold the line number of the data block, most significant byte first, and the third byte holds the length of the data (excluding the first three bytes). The 'CLS' could stand for 'Clear Screen', a simple enough routine to execute.

We *could* point the interpreter to this line number and allow it to look up the address of 'CLS' as done in Adventure games. But this method is too slow, especially when we want a speed critical high level language running at maximum speed. The interpreter must substitute the 'CLS' for a *token*.. This is a number representing the 'CLS' instruction. Why bother to do this? Replacing the instruc-

tion with a number immediately saves two bytes, and even more for longer instructions.

As stated, the language can operate more quickly and efficiently by searching a set of numbers instead of string decoding. Moreover, we can fit more instructions on each line. The maximum limit is currently 250 bytes per line, but this can be pushed up to 255, or even higher (up to 65535) by reserving two bytes for the line length instead of one.

A language like PILOT, which uses single letter instructions, requires no tokenisation or word interpretation at all. More sophisticated languages like BASIC or PASCAL requires definite, full length words like 'PRINT' and 'WRITE' to be stored in some compressed format.

To turn this program into a fully fledged language you need to: Modify the GLINE routine, probably by adding a CALL, that will read the contents of BUFFER and tokenise it (this will require a string decoder); Modify the LIST routine to list the full word, not the token itself; and add an 'R' (run) command in CTABLE. The run routine must scan the tokens and jump to the appropriate machine language subroutine. For 'CLS' this could be:

```
CLS     LD      HL,VDU          ;START OF VDU
        LD      DE,VDU+1
        LD      BC,LEN-1        ;LENGTH OF RAM
        LD      (HL),32         ;ASCII BLANK
        LDIR                    ;CLEAR
        LD      HL,PNT          ;BOTTOM LEFT
        LD      (LOC),HL        ;HOME CURSOR
        RET
```

This routine is straightforward enough routine for a simple command. A novice language is expected to handle strings, variables, decimal numbers, maths and other functions. Of course if you have come up with a computer language that does away with the need for all these things, but still solves the same problem, fantastic!

You may be thinking of a language for the expert which retains the same degree of power as machine code, but is easier to write, edit and debug. Or special

languages for special applications; business, education (PILOT is a good example) and entertainment are all possibilities. Many specialized languages exist for artificial intelligence and maths, but many more are lacking. Frankly, I don't see the point of having line numbers. It would be better to build an index table – a table with addresses pointing to labels – and CALL or 'GOTO' labels instead of line numbers. Nevertheless, if you want to develop an authentic version of a language like PASCAL for your machine you could not do without them.

Listing 6.0 serves to summarize much of what we have covered in the book so far. A complete keyboard debounce routine, scanning every key on the keyboard, including alpha-numerics, punctuation and other special symbols. The display and convert decimal routines. Block move calculations and ASCII display routines. A little over a half of the list is therefore made up of previously documented routines now put to the test – so don't let its size dishearten your potential understanding of it. The only sections requiring detailed analysis, for an understanding of record maintenance, are the insert, delete and list routines.

The program begins by clearing the VDU and displaying a message. After loading the appropriate registers it enters the execution control loop. The CBUFF routine clears the buffer where temporary data is kept before being placed in a line number. The SHOW routine acts as the character display routine. The GETK routine is the user input routine. It is terminated by an ASCII 13, 'carriage return' (this should be the ENTER or RETURN key on your computer).

The WORK routine converts the ASCII numbers into their numeric equivalents. It accepts the formats 'C nnnn' where 'C' is a one byte ASCII instruction (the choices are 'I', 'D' or 'L') and 'nnnn' is a number between 0-65535. It will also accept the format 'C nnnn,nnnn', where the second 'nnnn' is another number between 0-65535. This second number is not used in any routine, although it must still be a valid number for the entire command to be accepted. It has been included solely for the purpose of later expansion. The first number is held in the address LINE, the second in LINE1.

Once the keyboard has been read, the program expects the one letter command to be in the first byte of BUFFER. The contents of this address are loaded into the accumulator. The FKEY routine does a block compare and table search, ending with a jump to the corresponding routine, else an error message is displayed. Such a routine is documented in chapter four.

The LIST routine begins by CALLing LIMIT, which, by means of a subtract, compares the current address of the line number with the address of the last line number in memory. When the current line number is greater than the last line number (no overflow), all the lines have been listed and the program RETurns. The address of the line number in memory is loaded into DE, the requested line number

in HL. Had the operator typed 'L50', all the lines before 50 would be skipped by executing PASS. The routine calculates the location of the next line number by adding (IX + 2), which is the length of the data, to IX and adding another three bytes (two bytes for the line number, one byte for the length).

Suggestions for expanding this routine: Take advantage of LINE1 which holds the second address. The input 'L50,90' could list all lines between 50 and 90. Insert a compare after LIMIT is CALLed to do this. This can be done by subtracting the current line (in IX) from LINE1 and checking for an overflow. As well, a token conversion routine can be placed in PLP. This must look up the number in A, match a word, and display the whole word instead of the ASCII representation of the token.

The DELETE routine begins, like LIST, by loading IX with the first address of the memory holding the line numbers. The line number (located in IX and IX + 1) is loaded into DE and compared to HL (which holds the address in LINE). If a match is found, a jump is made to FOUND, else the routine calculates the address of the next line number by adding to IX the contents of (IX + 2) + 3. The routine, FOUND, deletes the desired line number by moving the rest of the line numbers down over the old record. Then TCODE, which holds the address of the very last byte used by a line number, is adjusted to compensate for the removed record. Examine diagram 6.3.

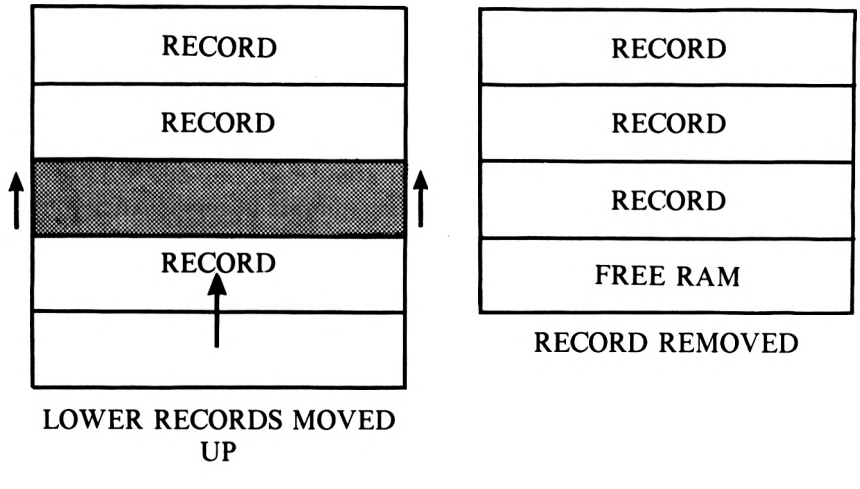


DIAGRAM 6.3 – DELETE A RECORD

The INSERT routine is fractionally more complex. Either the operator wants to insert a line number inside the file or at the end of the file. If it becomes necessary to insert in between used memory (for example: the operator inserts a line 50 when a 40 and 60 already exist), all memory after data block 60 must be moved up in memory to make room. Examine diagram 6.4. The DOIT routine makes space for the record and the BLOCK routine moves the line number data from the BUFFER into its newly made location. This may seem like a slow way of doing things. Actually the Z80 block move instructions are very fast, and there is very little speed difference working with a large or small file.

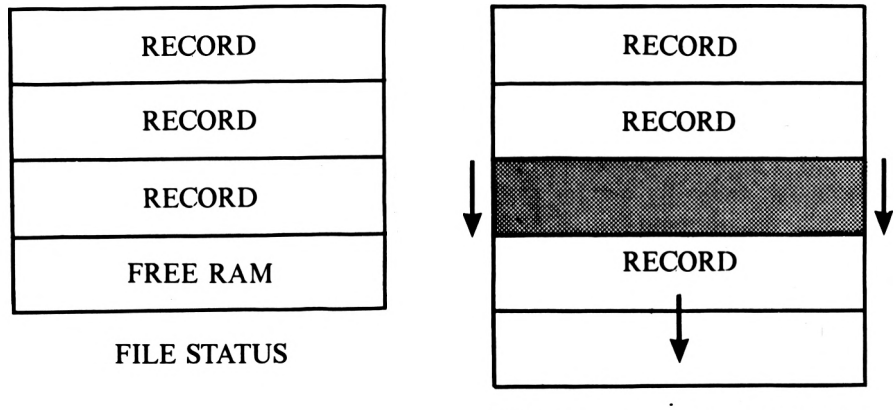


DIAGRAM 6.4 – INSERT A RECORD

Suggestions for improvement: The GLINE routine is the one that gets the information from the buffer and places it in line number memory. After you have typed '150' for example, the following will appear:

```
50 _
```

The GLINE routine displays the decimal number and allows you to type text which goes directly into the buffer. After entering a carriage return, the GLINE routine exits back to INSERT. In a completed language, GLINE would also handle the 'interpretation' of the English (or other ASCII) words into tokens. These tokens would then be moved into the piece of memory created by the DOIT routine. Although this routine will be left to your own devising, I do have a few suggestions. Move the tokens directly into the line number, rather than placing them back in the buffer (which will make the routine hopelessly confusing) or put them into a separate buffer (but 250 bytes must be reserved).

The routines GKEY and SHOW are machine dependant. Provided that you have an ASCII character set and are able to directly load VDU RAM with these characters, SHOW will require no modification. On the other hand GKEY, for memory mapped keyboard input, has been written *to be* modified. The following is a modification check-list when converting the keyboard decoder over to your computer:

- The debounce routine delay. A faster computer will require a longer delay.

- The number of bytes to be individually scanned for each row of keys. There is, naturally, a limit of eight keys to the byte – one key for each bit. This count is held in the E register.

- Whether the bits go 'high' or 'low' (set or reset) when a key is pressed. The JR NC instruction in PRESS would have to be changed to JR C.

- The BTAB routine holds the ASCII characters in the order of bits in each byte. If three bits of a byte correspond to keys, but the other four bits are ignored, leave four blank spaces in the table.

- The KTAB data table holding the memory-mapped keyboard addresses must be changed.

- If your character set is not standard ASCII you will have to modify the FIX routine to adjust upper and lower case automatically. The routine is not ASCII dependant, and will just as readily accept the EBCDIC or other standards by changing BTAB.

- If your keyboard is port addressed, other factors will have to be taken into consideration.

Note that no provision is made for lower case. Many computers do not have 'real' lower case letters, but have two sets of upper case or a set of inverse. Lower case is ignored by the program, although you can change this to suit your needs.

CHAPTER 7
– THE VICIOUS CIRCLE

This chapter is about writing programs to help you write better programs – a vicious circle. Many programmers specialize in writing software to assist other programmers. The 'utility' program usually helps debug, edit, study, write, or repair another program, whether this is in memory or on a mass storage device. We are concerned with programs in addressable memory, written in Z80 machine code, that, for the most part, don't work.

7.1 WRITING PROGRAMS TO WRITE PROGRAMS

The parallel of the golfers' hole-in-one for the programmer is the code that runs properly first-time. I have not, to-date, ever typed a substantial piece of Z80 code into an editor/assembler without needing to revise, modify, or rewrite some of it. Nor do I ever expect to. Humans make mistakes, but when programming a computer we have to go back and fix them. Worse still, we get to count them.

It is very tempting to give up a program (or for that matter, programming) when faced with debugging. The first sane thing to do is to buy a program that will help you find errors. It should at least 'single-step' and 'disassemble' memory (more on these terms later). If your computer lacks such luxuries, you should try writing one. Such a utility is called a *monitor.* A monitor is not the most difficult program to write, and a good project for a novice's first major work: The reason why this chapter is near the end of the book.

As stated, if such a program already exists, go out and buy it. It is better than knocking yourself out on a project that is unlikely to give you any profitable returns. Supposing there is no program available, or it is hopelessly inadequate, consider it your first – and most important – objective. For time saved in debugging alone, it's worth the effort. There might also be a commercial market for it, so write it professionally.

What used to be mandatory with a computer – an editor/assembler, may not appear until after the machine has become widely accepted. Yet to promote the computer, the manufacturer usually releases a few machine code programs as well. Possibly a simple business program or two and a few games. How were they written without an editor/assembler or even a utility program? A firm often writes a program on an established computer, which already has a complete complement of programming aids and an editor/assembler, then transfers the program over to the new computer by typing in the hexadecimal output of the assembly listing.

The hexadecimal input routine is either written in a high level language supplied in the computer's ROM, or possibly a very simple, ROM based, monitor. Special hardware techniques might also be used. Unless you are a whiz with a soldering iron, we can rule out the latter, or have access to an alternate computer, rule out the first possibility too.

Your only option is to write a hexadecimal input routine in a high level language. Construct a very simple monitor – with memory display and editing features, and then use this to write the rest of a more sophisticated monitor. Once complete, think about selling it, or if you are a ruthless entrepeneur, keep it for yourself and write software faster than the competition. This chapter deals with writing such a monitor, and editing your program in machine code form. This process is called *patching,* and is merely the process of typing over defective code with the (hopefully) correct instructions.

Without an editor/assembler your program will probably end up as one big, complex, confusing patch. Adding finishing touches and extra routines requires the insertion of CALLs over previous code, which must be retyped in the subroutine. Possibly you have almost completed a large program and you do not wish to reload a large piece of source code back into the editor/assembler to make a few small modifictions.

If you are trying to get just the right sound effect, or a perfect delay for your debounce routine, it becomes impractical to reload and save source over and over again for 'fine' adjustments. It's easier to load the program in memory, with the monitor, and jump between the monitor and your program until you have settled on the appropriate constants, variables and other data for the final version. For convenience sake write a routine that will jump back to your monitor on the press of a particular key. The final product will have this feature disabled. Preferably, write down all the modifications and touch-ups and load your source code in one last time. Once you have run your program several times in memory, registers and memory addresses are upset and hold strange values. A master copy outputted by an editor/assembler is the best copy.

All addresses and other words are stored in backward sequence in the Z80. Using an editor/assembler you needn't worry about this as the necessary calculations are made automatically. You would have noticed that a word stored in memory requires the DW pseudo-op rather than DB. This instruction specifies you want the number stored backwards in memory, otherwise it will be read backwards (LSD first) by an LD HL,(DATA) or similar instruction.

Likewise, relative jump addresses are also calculated by the assembler. When working with machine code these calculations have to be made manually. Once

again a programmer's calculator, or scientific calculator that handles base 16 conversions, is highly recommended.

Examine the instruction JP 5044H. This is held in memory as 0C3H, 44H and 50H. The '0C3H' is the code for JP, the two other bytes are in the expected reverse sequence. The CALL and LD instructions require this same reverse sequence as well. Such that LD HL,6050 would appear in memory as 21H, 50H and 60H.

Relative addressing is just as easy once you have the hang of it. The instruction JR $ (remember that '$' means the first byte of the current instruction) in machine code is 18H, 0FEH. The '18H' identifies JR, and the '0FEH' represents a jump of zero (actually negative two, as it jumps back to restart the instruction). To calculate the real $ add two. So that JR $+5 would appear in memory as 18H, and 03H. It is also possible to jump backwards in memory, since the JR instruction has a positive and negative range. The instruction 18H, 0FDH stands for JR $-1. The instruction 18H, 0C8H as JR $-54 and so on. Any number between 0FFH (although this number is very rarely used since it is a jump to the second byte of the JR instruction) and 07DH (125 – remember to add 2) is positive. Try calculating the relative addresses of these code sequences: 18H, 00H; 18H, 7EH; 18H, 7DH; 18H, FFH; 18H, 80H.

In an editor/assembler we can write:

```
LOOP    LD      HL,5000H
        OR      A
        JR      NZ,SKIP      ;$+4
        DEC     HL
        LD      A,(HL)
SKIP    EX      DE,HL
```

Moving the label up one instruction, obviously, recalculates the relative jump. This is only if we are working with source code, and not, 'disassembled' material. A disassembler is the opposite of an assembler, taking previously assembled code and displaying it in mnemonics form. A disassembler output would look like:

```
9000    21 00 50     LD     HL,5000H
9003    B7           OR     A
```

```
9004    20 02       JR    NZ,$+4     ;9008H
9006    2B          DEC   HL
9007    7E          LD    A,(HL)
9008    EB          EX    DE,HL
```

As illustrated, to calculate the relative address, subtract the address you want it to jump to from the $ of the JR, then subtract two.

To be able to work effectively in machine language, try and memorise the hexadecimal codes for the following instructions: LD HL,nn; LD BC,nn; LD DE,nn; JP nn; RET; CALL nn; CP n; LD (HL),n; JP Z,nn; LDIR; all of which you will need to use constantly.

7.2 SINGLE-STEPPING

The simplest debugging aid inserts 'breakpoints' in the routine you are testing. This is usually an RST that has been revectored into high RAM (if you're unsure about what I'm talking about, forget this method) and then back into your monitor, or, a JP nnnn is placed over a part of your program, again forcing it to jump back to your monitor. Basically, insert the jump or restart after the section of code you are suspect of, and then execute the program. And make sure the breakpoint has not been placed over any data used by the routine.

When it reaches the breakpoint, it jumps back to the monitor, where all register contents are displayed. Provided that you are able to spot anything out of place by these values a bug may be found. A true breakpoint is self-correcting. The bytes overwritten by the next instruction are saved away, and then restored in their original locations when you re-enter the monitor. Anything in ROM, obviously, is not breakpointable.

The code required to run such a routine is trivial, so we will move on to more sophisticated applications. Breakpointing is popular, and almost every monitor program has this feature, but I am one to question its worth. We would have to run through the routine instruction by instruction before we knew what values were expected, unless an obviously impossible value was RETurned. Subtle errors can easily be missed, especially when breakpointing.

Checking each value is long and laborious, even with a calculator, and we would probably spot the error studying each instruction anyway. Breakpointing does not

save very much debugging time. A better way is to 'single-step' the machine code. One way is to breakpoint automatically after each instruction, so we can observe the effects on every register, one step at a time. Doing this, we can follow along with the program, let the Z80 do all calculations for us, and simply confirm operation results.

Inserting a breakpoint after each instruction, brings us back to our previous problem: ROM memory can still not be single-stepped. Another approach is to move the instruction into a buffer and execute it there. The address after the buffer is where we would have wanted the program to jump anyway.

Of course such a routine requires a lot more work. The program must know the length of each instruction before it can be moved, and the jump addresses of RSTs, CALLs, JPs, JRs, and RETs have to be modified in case the program jumps out of monitor control.

The majority of instructions not beginning with an 0EDH, 0FDH or 0DDH are one byte in length. To determine the true length of each instruction we need two tables, one of the first bytes of two byte length instructions, and one of the first bytes of three byte length instructions. Block compares will sort through these tables. Of the instructions beginning with 0EDH, six have a length of three, the rest two. Of the 0FDH and 0DDH instructions, eleven are two bytes long, any third byte equalling 0CBH is three bytes, the remainder four. No match at all means the instruction must have a length of one. On entry, DE points to the first byte of the instruction, and on exit instruction length is in C:

```
LEN     LD      HL,TABLE0       ;2 BYTES LONG
        LD      BC,23           ;29 OF 'EM
        LD      A,(DE)          ;GET 1ST BYTE
        CPIR                    ;BLOCK COMPARE
        JR      Z,LEN2          ;FOUND IF LEN 2
        CP      0EDH            ;CHECK THIS,
        JR      Z,LEN5          ;INST. GROUP,
        CP      0DDH            ;AND THE NEXT
        JR      Z,LEN6          ;2
```

```
        CP      0FDH
        JR      Z,LEN6
        LD      HL,TABLE3   ;3 BYTES LONG
        LD      BC,26
        CPIR
        JR      Z,LEN3      ;YES, 3 BYTES
        LD      C,1         ;MUST BE 1 BYTE
        RET
LEN5    LD      HL,TABLE1   ;3 BYTES LONG?
        LD      BC,6
        INC     DE          ;2ND BYTE
        LD      A,(DE)
        CPIR
        JR      Z,LEN3      ;YES!
LEN2    LD      C,2         ;IS 2 BYTES
        RET
LEN3    LD      C,3         ;IS 3 BYTES
        RET
LEN6    LD      HL,TABLE2   ;2 BYTE LENGTH
        LD      BC,11
        INC     DE          ;2ND BYTE
        LD      A,(DE)
```

```
        CPIR
        JR      Z,LEN2          ;2 BYTES LONG
        INC     DE              ;3RD BYTE
        LD      A,(DE)
        CP      0CBH            ;IS IT 3 BYTES?
        JR      Z,LEN3
        LD      C,4             ;MUST BE 4
        RET
TABLE0  DB      6,0EH,10H,16H,18H,1EH,20H
        DB      26H,28H,2EH,30H,36H,38H
        DB      3EH,0C6H,0CBH,0CEH,0D3H
        DB      0DEH,0E6H,0EEH,0F6H,0FEH
TABLE1  DB      43H,4BH,53H,5BH,73H,7BH
TABLE2  DB      9,19H,23H,29H,2BH,39H,0E1H
        DB      0E3H,0E5H,0E9H,0F9H
TABLE3  DB      1,11H,21H,22H,2AH,31H,32H
        DB      3AH,0C2H,0C3H,0C4H,0CAH
        DB      0CCH,0CDH,0D2H,0D4H,0DAH
        DB      0DCH,0E2H,0E4H,0EAH,0ECH
        DB      0F2H,0F4H,0FAH,0FCH
```

Once we know the instruction length we can subdivide a single-stepping program into a number of sections. The first will move instructions into the buffer and adjust the 'fake' PC to point to the next instruction. To keep the program under

control at all times, the routine must scan for CALLs, JPs, JRs, RETs, and RSTs. If one is found, the instruction must be modified to jump or CALL a special routine.

Taking an example, how would such a program process a 'JP Z,3411H' . . . ? First we scan for all JP conditions. When matched to an op code, a jump is made to a routine that will save 3411H in another address, and change it to 'JP Z,GOHERE'. Now, if the zero flag is set, the JP is valid and GOHERE is executed – else it's ignored and we don't worry further about the instruction. Consequently the address 3411H is loaded into the 'fake PC'.

A CALL instruction is slightly more complex. The RETurn address of the CALL is PUSHed onto the stack, but since it was CALLed from the BUFFER, instead of its true origin, the value on the stack will be misleading. A routine must POP this off, and replace it with its 'true' RETurn address (calculated when the instruction was block moved into the BUFFER). Recovering the address saved away before (the start address of the subroutine), we place this in the 'fake' PC, save away the new location of our 'fake' stack pointer and RETurn.

Alternately, how would we go about single-stepping a RET? Replace the RET in the BUFFER with a conditional jump (for example; RET C would be replaced with a JP C). If the condition is true, we would POP the value off the 'fake' stack pointer, place it in the 'fake' PC and save away both registers before RETurning.

The routine must also check for JP (HL), JP (IX) and JP (IY) which are two bytes long and cannot be searched using a CPIR like the others. A 'HALT' instruction is taken as an endless loop (DEC PC). Both RETN and RETI are invalid instructions since interrupts are always disabled. However, if you are stubborn enough to think otherwise, this is how to check for interrupts:

```
LD      A,I

JP      PO,DISBL     ;INTR. DISABLED
```

The special RST instructions jump to specific memory locations, so a table look-up calculates the appropriate address. The RETurn address is still PUSHed on the stack like a CALL instruction.

Of all the instructions, JRs are the hardest to calculate. First we have to work out the relative offset, and then test conditions. It's not possible to change JRs into JPs by shifting bits, since there are eight conditional JP instructions, but only four JRs, so no correspondencing bit patterns (the next section discusses bit patterns in greater detail) exist. The program must therefore test JRs individually.

On exit, another routine, of your own devising, should display the new contents of all 'fake' registers, the flag conditions, the sections of memory they point to, the hexadecimal instruction code of the instruction, and possibly the disassembled instruction itself. Each time a key is pressed ('S' for single-step for example) Listing 7.0 should be CALLed. The program should also allow you to change register contents at any time.

```
                00100 ;MACHINE LANGUAGE SINGLE-STEPPING SUBROUTINE
                00110 ;ENTRY -->        C  = LENGTH OF INSTRUCTION
                00120 ;
                00130 ;EXIT  -->        INSTRUCTION INTERPRETIVELY EXECUTED &
                00140 ;                 REGISTER ADDRESSES CHANGED ACCORDING
                00150 ;                 TO OPERATION.
                00160 ;*CHANGE ACCORDING TO COMPUTER
                00170 ;*****************************
                00180 ;
9000            00190           ORG     9000H           ;*
9000 ED5B4192   00200 STEP      LD      DE,(PC1)        ;GET PC ADDRESS
9004 ED734792   00210           LD      (RETURN),SP     ;SAVE RETURN ADDRESS
9008 21AF90     00220           LD      HL,BUFFER       ;EXECUTION BUFFER
900B E5         00230           PUSH    HL
900C 0604       00240           LD      B,4             ;CLEAR BUFFER
900E 3600       00250 CLP       LD      (HL),0
9010 23         00260           INC     HL
9011 10FB       00270           DJNZ    CLP
9013 EB         00280           EX      DE,HL           ;SWAP SOURCE
9014 D1         00290           POP     DE              ;HL INTO DE
9015 0600       00300           LD      B,0
9017 EDB0       00310           LDIR                    ;MOVE INST. IN BUFFER
9019 224192     00320           LD      (PC1),HL        ;SAVE NEW PC
901C DD7E00     00330           LD      A,(IX)          ;NOW COMPARE
901F FE76       00340           CP      76H             ;IS IT "HALT"?
9021 CAE690     00350           JP      Z,SKIP          ;END INTERPRETATION?
9024 210292     00360           LD      HL,TABLE        ;= CALL, JR, JP, RST, RET
9027 010900     00370           LD      BC,9            ;9 CALL CONDITIONS
902A EDB1       00380           CPIR
902C CAEF90     00390           JP      Z,FIXCAL        ;FOUND A CALL?
902F 010900     00400           LD      BC,9            ;9 JP CONDITIONS
9032 EDB1       00410           CPIR
9034 CA3491     00420           JP      Z,FIXJP         ;FOUND A JP?
9037 010900     00430           LD      BC,9            ;9 RET CONDITIONS
903A EDB1       00440           CPIR
903C CA8F91     00450           JP      Z,FIXRET        ;FOUND A RET?
903F 010600     00460           LD      BC,6            ;5 JR & 1 DJNZ CONDITION
9042 EDB1       00470           CPIR
9044 CA5E91     00480           JP      Z,FIXJR         ;FOUND A JR OR DJNZ?
9047 010800     00490           LD      BC,8            ;8 RST INST.
904A EDB1       00500           CPIR
904C CACE91     00510           JP      Z,FIXRST        ;FOUND AN RST?
904F FEE9       00520           CP      0E9H            ;IS IT "JP (HL)"?
9051 CA4B91     00530           JP      Z,FIXHL
9054 FEDD       00540           CP      0DDH            ;IS IT ANOTHER JP?
9056 200B       00550           JR      NZ,SKIP2
9058 DD7E01     00560           LD      A,(IX+1)        ;CHECK 2ND BYTE
905B FEE9       00570           CP      0E9H            ;IS IT A "JP (IX)"?
905D CA5091     00580           JP      Z,FIXIX
9060 DD7E00     00590           LD      A,(IX)
9063 FEFD       00600 SKIP2     CP      0FDH            ;ANOTHER JP?
9065 200B       00610           JR      NZ,SKIP3
9067 DD7E01     00620           LD      A,(IX+1)        ;CHECK 2ND BYTE
906A FEE9       00630           CP      0E9H            ;IS IT A "JP (IY)"?
906C CA5591     00640           JP      Z,FIXIY
906F DD7E00     00650           LD      A,(IX)
9072 FEED       00660 SKIP3     CP      0EDH
9074 2009       00670           JR      NZ,CONT
9076 DD7E01     00680           LD      A,(IX+1)
```

```
9079 FE45      00690          CP      45H             ;IT IS A "RETN"?
907B C8        00700          RET     Z               ;FORGET IT--NO INTERRUPTS
907C FE4D      00710          CP      4DH             ;IT IS A "RETI"?
907E C8        00720          RET     Z               ;IGNORE AS WELL
               00730 ;
907F 2A3392    00740 CONT     LD      HL,(AF1)        ;LOAD ALL REGISTERS
9082 E5        00750          PUSH    HL
9083 F1        00760          POP     AF
9084 ED5B2D92  00770          LD      DE,(DE1)
9088 ED4B2F92  00780          LD      BC,(BC1)
908C 2A2B92    00790          LD      HL,(HL1)
908F DD2A3592  00800          LD      IX,(IX1)
9093 FD2A3792  00810          LD      IY,(IY1)
9097 D9        00820          EXX
9098 08        00830          EX      AF,AF'
9099 2A3992    00840          LD      HL,(XAF1)
909C E5        00850          PUSH    HL
909D F1        00860          POP     AF
909E 08        00870          EX      AF,AF'
909F 2A3F92    00880          LD      HL,(XHL1)
90A2 ED4B3B92  00890          LD      BC,(XBC1)
90A6 ED5B3D92  00900          LD      DE,(XDE1)
90AA D9        00910          EXX
90AB ED7B3192  00920          LD      SP,(SP1)
90AF 00        00930 BUFFER   DB      0,0,0,0         ;EXECUTE INSTRUCTION
     00 00 00
90B3 222B92    00940 SAVE     LD      (HL1),HL        ;SAVE ALL REGISTERS
90B6 ED733192  00950          LD      (SP1),SP
90BA ED532D92  00960          LD      (DE1),DE
90BE ED432F92  00970          LD      (BC1),BC
90C2 F5        00980          PUSH    AF
90C3 E1        00990          POP     HL
90C4 223392    01000          LD      (AF1),HL
90C7 08        01010          EX      AF,AF'
90C8 F5        01020          PUSH    AF
90C9 E1        01030          POP     HL
90CA 223992    01040          LD      (XAF1),HL
90CD DD223592  01050          LD      (IX1),IX
90D1 FD223792  01060          LD      (IY1),IY
90D5 D9        01070          EXX
90D6 223F92    01080          LD      (XHL1),HL
90D9 ED533D92  01090          LD      (XDE1),DE
90DD ED433B92  01100          LD      (XBC1),BC
90E1 ED7B4792  01110 FINISH   LD      SP,(RETURN)     ;GET CORRECT SP
90E5 C9        01120          RET                     ;DONE!
               01130 ;
               01140 ;SUBROUTINES FOR BRANCH AND HALT INSTRUCTIONS:
               01150 ;
90E6 2A4192    01160 SKIP     LD      HL,(PC1)        ;ENDLESS LOOP
90E9 2B        01170          DEC     HL              ;ON HALT
90EA 224192    01180          LD      (PC1),HL
90ED 18F2      01190          JR      FINISH
90EF CDEC91    01200 FIXCAL   CALL    LOADHL          ;CALL ADDRESS IN "JPADRESS"
90F2 224592    01210          LD      (JPADDR),HL     ;AND MODIFY CALL
90F5 21FE90    01220          LD      HL,GOHERE       ;NEW JP ADDRESS
90F8 CDF391    01230          CALL    SAVEHL          ;NOW LOAD IT
90FB C37F90    01240          JP      CONT            ;AND EXECUTE IT
90FE E1        01250 GOHERE   POP     HL              ;WRONG ADDRESS OFF STACK
90FF 2A4192    01260          LD      HL,(PC1)        ;PROPER RETURN ADDRESS
9102 E5        01270          PUSH    HL              ;PUSH IT
9103 2A4592    01280          LD      HL,(JPADDR)     ;AND REAL PC
9106 224192    01290          LD      (PC1),HL        ;IN ADDRESS
9109 ED733192  01300          LD      (SP1),SP        ;SAVE SP
910D 18D2      01310          JR      FINISH
               01320 ;
910F DD7E00    01330 FIXRET   LD      A,(IX)          ;CHECK RET
9112 FEC9      01340          CP      0C9H            ;STRAIGHTFORWARD RETURN?
9114 2810      01350          JR      Z,GORET         ;YES, GO TO GORET
9116 F6C2      01360          OR      0C2H            ;11000010B (SET 7,6,1)
9118 E6FA      01370          AND     0FAH            ;11111010B (RESET 0,2)
```

```
911A 212691    01380          LD      HL,GORET        ;"RET" NOW A "JP"
911D DD7700    01390          LD      (IX),A          ;SAVE IT
9120 CDF391    01400          CALL    SAVEHL          ;AND JP ADDRESS
9123 C37F90    01410          JP      CONT
9126 ED7B3192  01420 GORET    LD      SP,(SP1)
912A E1        01430          POP     HL              ;RECOVER "RET" ADDRESS
912B 224192    01440          LD      (PC1),HL        ;NEW PC
912E ED733192  01450          LD      (SP1),SP        ;ADJUST SP
9132 18AD      01460          JR      FINISH          ;BACK
               01470 ;
9134 CDEC91    01480 FIXJP    CALL    LOADHL          ;SAVE REAL JP IN "PC2"
9137 224392    01490          LD      (PC2),HL
913A 214391    01500          LD      HL,JPBACK       ;A NEW JP ADRESS
913D CDF391    01510          CALL    SAVEHL
9140 C37F90    01520          JP      CONT
9143 2A4392    01530 JPBACK   LD      HL,(PC2)        ;YES, JUMP WAS VALID, SO
9146 224192    01540          LD      (PC1),HL        ;SAVE IT AS NEW PC
9149 1896      01550          JR      FINISH
               01560 ;
914B 2A2B92    01570 FIXHL    LD      HL,(HL1)        ;"HL1" IS NEW PC
914E 1808      01580          JR      SAVEPC
9150 2A3592    01590 FIXIX    LD      HL,(IX1)        ;"IX1" IS NEW PC
9153 1803      01600          JR      SAVEPC
9155 2A3792    01610 FIXIY    LD      HL,(IY1)        ;"IY1" IS NEW PC
9158 224192    01620 SAVEPC   LD      (PC1),HL
915B C3E190    01630          JP      FINISH
               01640 ;
915E DD7E01    01650 FIXJR    LD      A,(IX+1)        ;CALCULATE RELATIVE
9161 2A4192    01660          LD      HL,(PC1)        ;OFFSET
9164 0600      01670          LD      B,0
9166 FEFE      01680          CP      0FEH            ;IS IT 0?
9168 3858      01690          JR      C,JR2           ;LESS THAN
916A 2808      01700          JR      Z,JR4           ;IF EQUAL, SKIP
916C FEFF      01710          CP      0FFH            ;IF 1, INC
916E 2002      01720          JR      NZ,JR3
9170 3E01      01730          LD      A,1             ;OFFSET IS 1
9172 4F        01740 JR3      LD      C,A             ;ADD OFFSET
9173 09        01750          ADD     HL,BC
9174 224392    01760 JR4      LD      (PC2),HL        ;SAVE IT
9177 DD7E00    01770          LD      A,(IX)          ;GET TYPE OF JR OR DJNZ
917A 2A3392    01780          LD      HL,(AF1)        ;AND FLAGS
917D E5        01790          PUSH    HL              ;SAVE "AF1" ON STACK
917E FE10      01800          CP      10H             ;IS IT A "DJNZ"?
9180 282C      01810          JR      Z,FIXDJZ
9182 FE38      01820          CP      38H             ;IS IT "JR C"?
9184 2810      01830          JR      Z,CARRY
9186 FE30      01840          CP      30H             ;IS IT "JR NC"?
9188 2812      01850          JR      Z,NCARRY
918A FE20      01860          CP      20H             ;IS IT "JR NZ"?
918C 2814      01870          JR      Z,NZERO
918E FE28      01880          CP      28H             ;IS IT "JR Z"?
9190 2816      01890          JR      Z,ZERO
9192 E1        01900          POP     HL              ;JUST "JR"
9193 C34391    01910 BACK     JP      JPBACK
9196 F1        01920 CARRY    POP     AF              ;CHECK IF CONDITIONS
9197 38FA      01930          JR      C,BACK          ;ARE VALID BY CHECKING
9199 C3E190    01940          JP      FINISH          ;FLAG
919C F1        01950 NCARRY   POP     AF
919D 30F4      01960          JR      NC,BACK
919F C3E190    01970          JP      FINISH
91A2 F1        01980 NZERO    POP     AF
91A3 20EE      01990          JR      NZ,BACK
91A5 C3E190    02000          JP      FINISH
91A8 F1        02010 ZERO     POP     AF
91A9 28E8      02020          JR      Z,BACK
91AB C3E190    02030          JP      FINISH
91AE ED4B2F92  02040 FIXDJZ   LD      BC,(BC1)        ;GET B REGISTER
91B2 F1        02050          POP     AF              ;GET FLAGS
91B3 05        02060          DEC     B               ;JUST LIKE DJNZ
91B4 F5        02070          PUSH    AF              ;SAVE NEW FLAG
```

```
91B5 E1          02080          POP     HL
91B6 223392      02090          LD      (AF1),HL
91B9 ED432F92    02100          LD      (BC1),BC        ;SAVE IT
91BD 20D4        02110          JR      NZ,BACK         ;LOOP BACK?
91BF C3E190      02120          JP      FINISH          ;NO LOOP
                 02130 ;
91C2 FE7F        02140 JR2      CP      7FH             ;THIS CALCULATES
91C4 38AC        02150          JR      C,JR3
91C6 ED44        02160          NEG                     ;NEGATE A
91C8 4F          02170          LD      C,A             ;NOW SUBTRACT A FROM "PC1"
91C9 B7          02180          OR      A
91CA ED42        02190          SBC     HL,BC
91CC 18A6        02200          JR      JR4             ;GOT NEW ADDRESS
                 02210 ;
91CE 2A4192      02220 FIXRST   LD      HL,(PC1)        ;CALL RST ADDRESS
91D1 ED7B3192    02230          LD      SP,(SP1)
91D5 E5          02240          PUSH    HL              ;PUSH RETURN ADDRESS
91D6 ED733192    02250          LD      (SP1),SP
91DA 21F991      02260          LD      HL,RTABLE-1     ;GET NEW PC
91DD 41          02270          LD      B,C
91DE 04          02280          INC     B
91DF 23          02290 RST1     INC     HL
91E0 10FD        02300          DJNZ    RST1
91E2 4E          02310          LD      C,(HL)          ;NEW PC IN HL
91E3 0600        02320          LD      B,0
91E5 ED434192    02330          LD      (PC1),BC        ;SAVE IT
91E9 C3E190      02340          JP      FINISH
                 02350 ;
91EC DD6602      02360 LOADHL   LD      H,(IX+2)
91EF DD6E01      02370          LD      L,(IX+1)
91F2 C9          02380          RET
                 02390 ;
91F3 DD7402      02400 SAVEHL   LD      (IX+2),H
91F6 DD7501      02410          LD      (IX+1),L
91F9 C9          02420          RET
                 02430 ;
                 02440 ;"RTABLE" HOLDS PC ADDRESS FOR RST INST:
91FA 38          02450 RTABLE   DB      38H,30H,28H,20H,18H,10H,8,0
     30 28 20 18 10 08 00
                 02460 ;
                 02470 ;"TABLE" HOLDS ALL BRANCH AND HALT INST:
9202 DC          02480 TABLE    DB      0DCH,0FCH,0D4H,0CDH,0C4H,0F4H,0ECH,0E4H
     FC D4 CD C4 F4 EC E4
920A CC          02490          DB      0CCH,0DAH,0FAH,0D2H,0C3H,0F2H,0C2H,0EAH
     DA FA D2 C3 F2 C2 EA
9212 E2          02500          DB      0E2H,0CAH,0C9H,0D8H,0F8H,0D0H,0C0H,0F0H
     CA C9 D8 F8 D0 C0 F0
921A E8          02510          DB      0E8H,0E0H,0C8H,38H,30H,20H,28H,18H,10H
     E0 C8 38 30 20 28 18 10
9223 C7          02520          DB      0C7H,0CFH,0D7H,0DFH,0E7H,0EFH,0F7H,0FFH
     CF D7 DF E7 EF F7 FF
                 02530 ;
                 02540 ;VARIABLES:
                 02550 ;
922B 0000        02560 HL1      DW      0               ;GENERAL REGISTERS
922D 0000        02570 DE1      DW      0
922F 0000        02580 BC1      DW      0
9231 0000        02590 SP1      DW      0               ;STACK POINTER
9233 0000        02600 AF1      DW      0               ;ACCUMULATOR & FLAGS
9235 0000        02610 IX1      DW      0               ;INDEX
9237 0000        02620 IY1      DW      0               ;INDEX
9239 0000        02630 XAF1     DW      0               ;ALTERNATE ACC. & FLAGS
923B 0000        02640 XBC1     DW      0               ;ALTERNATE REGISTERS
923D 0000        02650 XDE1     DW      0
923F 0000        02660 XHL1     DW      0
9241 0000        02670 PC1      DW      0               ;PROGRAM COUNTER
9243 0000        02680 PC2      DW      0               ;RESERVE PROGRAM COUNTER
9245 0000        02690 JPADDR   DW      0               ;STORAGE
9247 0000        02700 RETURN   DW      0               ;LOCATION OF "REAL" SP
9000             02710          END     STEP
00000 Total errors
```

Single-stepping, undoubtably, is the best way to learn about the Z80, experiment, destroy myths and master machine language. Limitations are: The single-stepper executes each instruction many times slower – it has many more things to do – and therefore cannot be used for real-time applications or programs that use disk and other speed critical I/O. Nor can it be used to test interrupt routines – unless of course you are determined to modify the program, but remember that the program being single-stepped is running at a different speed and may confuse interrupt handling. In the long run however, this is less than one per cent of all programming.

7.3 DISASSEMBLING

A disassembler should be a standard feature of any monitor. It makes machine code readable and at least bearable to edit. It also lets you study other people's coding for clues and techniques. Disassembling is not easy. Block comparing well over 600 separate instructions is not practical. There are however 'tricks' that allow subroutines to disassemble whole sections of instruction set. These are bit patterns.

Were you to examine the hexadecimal representation of instruction code, you would probably find nothing to associate a RET C with a RET M. In binary however definite patterns exist. For example: 11**011**000B represents RET C, 11**111**000B, RET M. Three bits can be organised into eight different patterns – one for each conditions.

Bit pattern	**Condition**
000	NZ
001	Z
010	NC
011	C
100	PO
101	PE
110	P
111	M

And apply to JPs as well as CALLs. A JP Z in binary is 11**001**010B, a CALL Z is 11**001**100B. To change a JP Z instruction into a CALL Z just SET 2,r and RES 1,r. The following table gives the bit patterns of all single registers and the contents of HL.

Bit pattern	Register
000	B
001	C
010	D
011	E
100	H
101	L
110	(HL)
111	A

In a single register 'LD' bit 7 is reset, bit 6 set. The instruction LD E,H in binary is 01**011100**B or 5CH. Now it becomes obvious why the Z80 has only six readily useable general purpose registers, an accumulator and one special addressing mode. You just run out of bits. Further registers would require another identification byte. Other bit patterns include:

Bit pattern	Register pair
00	BC
01	DE
10	HL
11	SP

As would be used in a ADD HL,?? instruction.

Bit pattern	Special registers
00	BC
01	DE
10	IX (or IY)
11	SP

As would be used in a ADD IX,?? instruction.

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

As would be used in the instruction SET 5,A.

Bit pattern	Restart address
000	00H
001	08H
010	10H
011	18H
100	20H
101	28H
110	30H
111	38H

As would be used in the RST 28H instruction.

A disassembler should be broken up into manageable routines. Each routine should be designed to disassemble one particular instruction format. In some cases a small routine can handle the disassembly of over a hundred instructions, while you will have to make separate compares for instructions that have nothing in common with others. Examples like HALT, DJNZ, EXX, NEG and SCF. The 8-bit loads can be handled by one routine. All told, a well written disassembler should be about three and a half Ks long. The following code is an example disassembler subroutine. It decodes all op codes between 40H and 7FH of the 8-bit load group, totalling 63 instructions. Preliminary code must identify instructions between these ranges. On entry the accumulator holds the contents of IX, which points to the first byte of the instruction to be disassembled.

```
;ENTRY -->    IX POINTS TO INSTRUCTION
;             A = HOLDS CONTENTS OF IX
;EXIT  -->    BC = 1ST & 2ND OPERANDS
;             HL = ASCII OP CODE
;
DECODE  CP    40H
        JR    C,PASS
        CP    80H
        JR    C,LDSBIT
        . . .
```

```
LD8BIT  CP    76H          ;HALT INST.
        JR    Z,STOP       ;REST ARE LD'S
        LD    HL,LD        ;'LD' ASCII
        PUSH  AF           ;SAVE BYTE
        AND   7            ;00000111B
        LD    C,A          ;2ND OPERAND
        POP   AF
        RRCA               ;1ST OPERAND
        RRCA               ;INTO 2ND
        RRCA               ;POSITION
        AND   7            ;CLEAR UNWANTED
        LD    B,A          ;1ST OPERAND
        RET
STOP    LD    HL,HALT      ;'HALT' ASCII
        LD    BC,1010H     ;NO OPERANDS
        RET

HALT    DB    'HALT.'
LD      DB    'LD.'
```

Initially HL is pointed to the first byte of the ASCII representation of the op code. The display routine will then show this mnemonic on the VDU. The bit patterns are shifted (or already are) to the leftmost part of the byte, making it possible to assign each bit pattern a number. The AND 7 masks out all unwanted bits. The number 00000111B, which stands for the A register, is now seven in decimal. A

display routine must show this information visually. Note: the display routine will become progressively more complex as it handles more and more instruction types. The display routine listed in this book is adequate for the few hundred 'demonstration' instructions being disassembled.

A suitable display routine follows:

```
;DISPLAY DISSASSEMBLED INSTRUCTION
;ENTRY --> HL = ASCII OP CODE
;      --> BC = 1ST & 2ND OPERANDS
;
DISP    CALL PRINT        ;SHOW OPCODE
        CALL TAB          ;SHOW A TAB
        CALL WORK         ;SHOW OPERANDS
        CP   32           ;IS IT BLANK?
        JR   Z,NODISP     ;SKIP IT THEN,
        LD   A,','        ;SHOW A COMMA
        CALL SHOW
NODISP  LD   B,C          ;2ND OPERAND
        CALL WORK
        RET
WORK    LD   HL,OPTAB-1   ;TABLE OF ASCII
        CP   6            ;IS IT A (HL)?,
        JR   Z,FHL        ;THEN PASS
        INC  B
LOOP    INC  HL           ;NEXT ASCII
```

```
        DJNZ  LOOP          ;COUNT DOWN
        LD    A,(HL)        ;GOT OPERAND
        CALL  SHOW          ;DISPLAY IT
        RET
PHL     LD    HL,DHL        ;A (HL)
        CALL  PRINT
        RET
PRINT   CP    '.'           ;END OF PRINT?
        RET   Z
        CALL  SHOW          ;DISPLAY IT
        INC   HL            ;NEXT BYTE
        LD    A,(HL)        ;IN A
        JR    PRINT         ;AGAIN
        RET

DHL     DB    '(HL).'
OPTAB   DB    'BCDEHLA01234567 '
```

The SHOW routine displays a character on the VDU. The TAB routine tabs to the next column, where operands will be displayed.

The routine DISP gets an ASCII character out of OPTAB according to the bits in B and C. A value of 16 in B or C will display a blank – no operands – for instructions without them.

The following code disassembles all BIT, SET and RES instructions, and requires the same parameters as the former:

```
;THIS CONTINUES AFTER DECODE:
        CP    0CBH          ;BIT/SET/RES
        RET   NZ            ;NOTHING ELSE
        LD    A,(IX+1)      ;NEXT BYTE
        CP    40H           ;TO LOW
        RET   C
        CP    80H
        JR    C,BITS        ;DECODE BIT
        CP    0C0H          ;DECODE RES
        JR    C,RESET
        LD    HL,SET        ;GOT SET
        JR    TEST
RESET   LD    HL,RES        ;GOT RES
        JR    TEST
BITS    LD    HL,BIT        ;GOT BIT
TEST    PUSH  AF            ;SAVE IT
        AND   7             ;MASK OUT
        LD    C,A           ;2ND OPERAND
        POP   AF            ;RECOVER
        RRCA                ;SHIFT INTO,
        RRCA                ;BITS 0-3
        RRCA
```

```
        AND   7             ;CLEAR THE REST
        ADD   A,7           ;POINT TO NO.'S
        LD    B,A
        RET

SET     DB    'SET.'
RES     DB    'RES.'
BIT     DB    'BIT.'
```

The only essential difference between this routine and LD8BIT is that seven is added to the first operand, pointing it to a number instead of a letter. Otherwise, in the completed disassembler, the shift and mask instructions would be placed in a subroutine.

Remember too that all bit instructions are in the format: 'BIT n,r', where n is a number between 0-7 and r is a register. Combined, these two routines disassemble 254 of the odd 600 plus, Z80 instruction set. There are many other problems you will have to overcome, but a thorough inspection of these routines will enable you to write the completed program.

CHAPTER 8
– SPECIAL APPLICATIONS

This chapter deals with the other, equally important, aspects of machine language programming. Special effects, undocumented instruction code, and then the process of developing, documenting, and *finally,* marketing your finished brain-child.

8.1 SOUND

Many computers are equipped with special sound synthesizer circuit boards to which the programmer only needs to pass parameters to. The sound generator will then beep, zap or ping without the programmer worrying further. Unfortunately, many more computers lack this feature. Moreover, a simple sound synthesizer is far from a *sound digitalizer.* Sound effect generation is machine dependant, so no code in this section is necessarily ideal for your requirements, but it can be modified to suit.

A computer must at least have a cassette interface to be defined as sound capable. The auxiliary plug (the one that must be plugged into the cassette recorder during the saving of code) must also fit into a speaker-amplifier. If you have ever played a software tape 'out loud' instead of into your computer, you would have noticed a high pitched screech. Every minute sound is a small magnetic impulse that the computer reads. Each small screech, only lasting a small fraction of a second, represents a bit. If there is no screech at an expected time (determined by the *baud* rate) the bit is taken as reset.

The tape is checked eight times for every byte. What is the relevance of this? By controlling the bits to output to tape, or in the case of sound – the speaker-amplifier, we can vary the frequency or 'pitch'. In effect, we have sound effects.

A cassette interface is normally controlled by sending bits to a specified port. This will enable the programmer to write information, turn the cassette recorder on or off, by way of a small relay (most computers have them), and also read the port for incoming data. To make a sound, we have to do the following: 1. Turn on the cassette recorder motor. 2. Send a bit through the port – our piece of 'sound'.

With the first bit gone, the cassette motor will cut out again, so we have to reactivate it. The following presumptions are made for the example: a 'set bit 0' will instruct the motor to turn on, and a 'set bit 1' is an instruction to write a bit to tape. The actual bits used will depend on your computer.

```
SOUND   LD      A,1             ;ACTIVATE
LOOP    XOR     3               ;SWITCH 1/2
        OUT     (RECORD),A      ;OUT RECORDER
        DJNZ    LOOP            ;REPEAT B TIMES
```

The number of times we switch bits will determine the frequency, the number of times we repeat the frequency will determine the duration.

```
SOUND   LD      D,(DURTN)
        LD      E,(FREQ)
        LD      A,1
LOOP    XOR     3
        OUT     (RECORD),A
        LD      B,E
FREQ1   DJNZ    FREQ1
        DEC     D
        JR      NZ,LOOP
```

While a simple beep, buzz, click, ding, pop, zap, or ping will do fine for most applications, it is far from 'real world' sound. The digital recording of sound is possible using a computer, by reading sound bits instead of software. But in practice, voices are the only sounds worth saving in this form. Microcomputers were not built with the intention of sound digitalization in mind, and so most systems have only marginally good sound reproduction.

A recorded voice is one of the few sounds that remains comprehensible once played back, and even this usually sounds 'metallic' or 'robotic'. There are also serious sound digitalizing drawbacks. Besides the poor reproduction you can expect (yet ideally suited to games) it uses tremendous amounts of memory. A single word can take up about 1K of memory – so make it important! The actual amount of memory required to store each word will depend on the cassette interface. Poor quality ones will need more memory to store a more 'detailed' voice. Better interfaces can store less detail. The more the detail, the better the reproduction.

Obviously games benefit from voice sound. A 'great shot!' and 'alert!' and a 'great game!' add flavour to computer games. A 64K system could hold the letters of the alphabet to enable word processing for the blind. Yet sound effects, and especially voice sound effects, have to be used in a program discretely. The computer cannot produce sound and continue to execute the rest of the program simultaneously. The duration has to be kept short, or CALLed for short periods between routines.

The following is an example of a voice digitalizer. The label MEM stands for the first byte of free RAM to be used, KEY scans for a key to be pressed and terminates the input if equal to STOP, BLANK is the value returned when no information is being sent to the port (this is not necessarily zero), and RECORD is once again the cassette interface port. The larger the number in DELAY, the greater the pause and the less memory used. A number too large may yield poor play back, a number too small, may use up too much memory. You will have to test the program yourself and find the best compromise.

```
VOICE   LD      HL,MEM
VOICE1  LD      C,8             ;NO. OF BITS
        XOR     A               ;CLEAR PORT
        OUT     (RECORD),A
        IN      A,(RECORD)      ;GET BYTE
        CP      BLANK           ;ANY BITS?
        JR      Z,SKIP
        SET     0,(HL)          ;A SOUND
        JR      VOICE2
SKIP    RES     0,(HL)          ;NO SOUND
VOICE2  RRC     (HL)            ;MOVE NEXT BIT
        LD      B,DELAY         ;PAUSE
LOOP    DJNZ    DELAY
        DEC     C               ;BITS LEFT?
```

```
        JR      NZ,VOICE1
        INC     HL              ;NEXT BYTE
        LD      A,(KEY)         ;KEYBOARD
        CP      STOP            ;HALT KEY
        RET     Z
        JR      VOICE
```

To play back (keeping in mind that you will have to modify this according to the bits used in your cassette interface):

```
PLAY    LD      HL,MEM          ;START
PLAY1   LD      C,8             ;8 BITS
PLAY2   LD      A,(HL)          ;GET BYTE
        AND     1               ;00000001B MASK
        INC     A               ;FOR CASSETTE
        OUT     (RECORD),A      ;BUZZ IT
        LD      B,DELAY         ;PLAYBACK DELAY
DELAY   DJNZ    DELAY           ;VARY TO TASTE
        RRC     (HL)            ;NEXT BIT
        DEC     C               ;ALL BITS?
        JR      NZ,PLAY2
        INC     HL              ;NEXT BYTE
        LD      A,(KEY)         ;STOP?
        CP      STOP
```

```
RET     Z

JR      PLAY1
```

At this point I can only wish you good luck with your experimenting. You could try varying the speed of play back on a reasonably clear recording to simulate different voices (a man's and a woman's for example). If you already have an internal speaker in your computer, you must find some way of playing the voice effects on this instead.

A word of caution: Some computers, because of appallingly bad cassette I/O, have been modified for more stable cassette operation. This may lock the baud into a specific rate, making it pointless to adjust for better results. It may also make it impossible to input (play back remains unaffected) comprehensible voice sound effects, unless this feature is disabled.

8.2 UNDOCUMENTED Z80

There are a number of special Z80 instructions that are not normally documented in elementary texts, since Zilog makes no reference to them. Understandably most editor/assembler programs will not accept them, at least not in mnemonic form, although many disassemblers presently on the market recognise these byte configurations. Programmers who have discovered some have given them odd mnemonic abbreviations (many have more in common with other microprocessors), so I have tried to clear up this confusion using symbolics as consistent as possible with the instruction set.

I have worked with many Z80's but I have never worked with one that did not accept these special instructions. This unfortunately is not a guarantee that they will work for you. The majority of Z80 computers seem to work with them. Nevertheless Zilog never tests these instructions for accuracy, so it is always possible that some versions of the chip are partially faulty, but are installed in computers regardless. For these reasons the instructions should never be incorporated in software you intend selling commercially.

Why didn't Zilog document these instructions? We could speculate that they were accidentally left out of documentation when released, and the firm made no later effort to correct the omission. Very possibly because they were not aware of them themselves. One could, unconvincingly, argue that they are of only limited use. I disagree. The instructions are all very useful, and it is obvious, in at least one case, that Zilog made a very definite mistake in not documenting what should have been a standard operation.

The 16-bit index registers can serve a dual purpose as four general purpose 8-bit registers. They do, like all index register operations, require an extra byte to be accessed. The first is a compulsory '0DDH' (IX) or '0FDH' (IY) identification byte. Selecting a mnemonic is not straightforward when IX and IY start with the same initial, but because of other similarities (explained in a moment) they have been given an 'L' or 'H' prefix, standing for 'low' order register (least significant byte), and 'high' order register (most significant byte). Examine these instructions:

```
LD      IX,5000H

LD      A,HX            ;HOLDS 50H

LD      B,LX            ;HOLDS 00H
```

Which helps to explain the mnemonics used. It is permissible to load any general purpose 8-bit register into HX, LX, HY and LY, add their contents to the accumulator with or without carry, load them with direct data, subtract their contents from the accumulator with or without carry, increment or decrement them, compare them and logically OR, XOR or AND them to the accumulator. Overall they can be manipulated in any standard way.

We can 'trick' the processor into accepting these instructions by adding the prefix byte before a standard Z80 instruction. To use them in an editor/assembler:

```
DB      0FDH            ;IY PREFIX

LD      A,L             ;LSB OF IY IN A
```

which is:

```
LD      A,LY
```

As you can see it is convenient to use the 'H' and 'L' prefix since this automatically corresponds to the H and L registers. For example:

```
DB      0DDH            ;IX PREFIX

LD      H,255           ;MSB OF IX IN A
```

is:

```
LD      HX,255
```

The mnemonic and instruction codes, as close to Zilog format as possible are as follows:

OBJECT CODE	SOURCE MNEMONIC		OBJECT CODE	SOURCE MNEMONIC	
DD67	LD	HX,A	DD6F	LD	LX,A
DD60	LD	HX,B	DD68	LD	LX,B
DD61	LD	HX,C	DD69	LD	LX,C
DD62	LD	HX,D	DD6A	LD	LX,D
DD63	LD	HX,E	DD6B	LD	LX,E
FD67	LD	HY,A	FD6F	LD	LY,A
FD60	LD	HY,B	FD68	LD	LY,B
FD61	LD	HY,C	FD69	LD	LY,C
FD62	LD	HY,D	FD6A	LD	LY,D
FD63	LD	HY,E	FD6B	LD	LY,E
DD26	LD	HX,n	DD2E	LD	LX,n
FD26	LD	HY,n	FD2E	LD	LY,n
DD7C	LD	A,HX	DD7D	LD	A,LX
DD44	LD	B,HX	DD45	LD	B,LX
DD4C	LD	C,HX	DD4D	LD	C,LX
DD54	LD	D,HX	DD55	LD	D,LX
DD5C	LD	E,HX	D05D	LD	E,LX

FD7C	LD	A,HY	FD7D	LD	A,LY
FD44	LD	B,HY	FD45	LD	B,LY
FD4C	LD	C,HY	FD4D	LD	C,LY
FD54	LD	D,HY	FD55	LD	D,LY
FD5C	LD	E,HY	FD5D	LD	E,LY
DD65	LD	HX,LX	DD6C	LD	LX,HX
FD65	LD	HY,LY	FD6C	LD	LY,HY
DD24	INC	HX	DD2C	INC	LY
FD24	INC	HY	FD2C	INC	LX
DD25	DEC	HX	DD2D	INC	LX
FD25	DEC	HY	FD2D	INC	LY
DD84	ADD	A,HX	DD85	ADD	A,LX
FD84	ADD	A,HY	FD85	ADD	A,LY
DD8C	ADC	A,HX	DD8D	ADC	A,LX
FD8C	ADC	A,HY	FD8D	ADC	A,LY

DD94	SUB	HX	DD95	SUB	LX
FD94	SUB	HY	FD95	SUB	LY
DD9C	SBC	A,HX	DD9D	SBC	A,LX
FD9C	SBC	A,HY	FD9D	SBC	A,LY
DDBC	CP	HX	DDBD	CP	LX
FDBC	CP	HY	FDBD	CP	LY
DDA4	AND	HX	DDA5	AND	LX
FDA4	AND	HY	FDA5	AND	LY
DDB4	OR	HX	DDB5	OR	LX
FDB4	OR	HY	FDB5	OR	LY
DDAC	XOR	HX	DDAD	XOR	LX
FDAC	XOR	HY	FDAD	XOR	LY

Had they been properly documented, life would have been much easier for the programmer.

There exists a left shift instruction, which was accidentally missed by Zilog – it even has a right shift equivalent. The documented shift instructions include; SLA, SRA and SRL. A shift right-arithmetic, shift right-logical, and shift left-arithmetic but no shift left-logical. This instruction has an '0CBH' prefix and is accordingly named 'SLL'. Its formats are:

OBJECT CODE	SOURCE MNEMONIC	
CB37	SLL	A
CB30	SLL	B
CB31	SLL	C
CB32	SLL	D
CB33	SLL	E
CB34	SLL	H
CB35	SLL	L
CB36	SLL	(HL)

The instruction is almost identical to SLA, except a '1' bit is placed in bit 0 instead of '0'. The following:

```
SLA     A
SET     0,A
```

Can be replaced by:

```
SLL     A               ;DB 0CBH,37H
```

There are many other code sequences which at present do nothing (unless you consider a two byte NOP an instruction). No doubt they will be used when more sophisticated versions of the chip appear.

8.3 WRITING COMMERCIAL SOFTWARE

A competent Z80 programmer can produce commercially viable work, but careful thought and a lot of determination is required. I suspect that many of my readers had the thought somewhere in the back of their minds when they bought this book. Chances are you have a market or particular computer in mind. This could be either for the Z80 range, or possibly, another computer that incorporates a Z80 co-processor, usually as an optional extra.

By fate, or, the limits of your budget, your most likely choice is the computer you get your hands on. Since Z80's are all essentially similar, it's possible to rewrite the same program for many different computers. The true professional starts writing with this intention in mind. Unfortunately, an excellent program can be of little worth on one Z80, while invaluable on another. If you write to sell, you must know why customers want your product.

A good business program sells because it's 'user friendly'. The businessman who bought his computer to save time, is not going to appreciate a program that is hard to use or overly complex. This is not to say simple is best; it is not. But, a powerful, sophisticated program does much of the work automatically; with the least fuss. Advanced features in word processing or database management are often selling points, but these extras should be invisible to the user, unless he or she has need for them. Then the press of the appropriate key gets the normally bypassed function.

Only in exceptional circumstances should a program ask questions. Query a file name when it comes time to save a file to disk, or, you need the name for a field in a database, or, you want to double-check a request because it might do something extremely important (or dangerous, like delete a record). On a word processor for example, there should be no need to ask for page lengths or if justification is required. The operator should adjust these variables when the need requires it. If the layman does not know the answer to the question, your program or documentation has failed.

You can expect more patience out of hobbyists, mainly because they know more about their computer in general. For the businessman the whole procedure has to be spelled out. Anything too brief can be ambiguous, anything too complex, confusing.

Take into account the number of steps needed to get to the 'heart' of your program. A good software package, where possible, should have two: insert disk and turn on computer. This is called a 'turn-key' system.

Where possible, display information in menu form. Like the following:

[A]DD INFORMATION

[S]ORT INFORMATION

[R]EMOVE INFORMATION

[D]ISPLAY INFORMATION

Which does not require the operator to remember any special keys. If you're unable to display information in menu form try and include a help command. Reassure the operator that pressing the 'H' (or whatever key) will help in case of trouble. A help routine can either give a menu or summary of commands.

Many programmers dislike giving helpful hints in the program because it encourages 'piracy' – giving a friend or associate an unauthorized copy. If you are worried about getting ripped off, reconsider commercial software writing. Money lost is money lost. You can only make it by giving your customer your complete support.

Be careful and consistent when choosing keys. Always try and have an override key. If there is an ESCAPE key, use that. Otherwise use BREAK, or CLEAR or STOP. If the operator can escape from a routine and return to the menu, getting out of trouble is easy. Never force a function to be carried out when there is the possibility of accidentally pressing the wrong key, or even a change of mind.

When assigning keys specific functions, such as in a word processing, try and make them as easy to remember as possible. To remove text, use the 'C' key to stand for 'change' or the 'D' for 'delete'. If the operator is likely to be familiar with typing terms, use 'O' for 'overtype'. Better still, read through typing books to see what format the typist is probably familiar with. Key selection is always an area of criticism, some argue that it's better to have important keys in convenient locations so the touch typist can save time. To avoid this disagreement altogether, allow the operator to redefine the keyboard.

To destroy a file or remove all the text in memory (which could be fatal if the operator does it accidentally), always follow up with a small question like SURE? (reply: yes/no). Specify the reply you expect as well. A person unfamiliar with your program could type 'OK' or 'positive' to a yes/no question. It has happened.

Never be fussy. Don't expect the operator to follow rigid restraints just because you are lazy. If the operator types in '$1,000' when your documention clearly stated just '1000', ignore it. Look for the dollar sign and commas, and write a routine to remove them. Some habits are hard to break, especially things like commas

between numerals. This lack of thought on the part of the programmer – for little things like these – is where most criticism is levelled. You will be surprised how magazine reviewers may endorse your product, simply because you showed a little consideration. Allow for many different date formats as well. Accept the date 11/11/84 just as readily as you accept 11-11-84 or 11:11:84. If you want to, convert it to the way you like, once it has been typed in. Remember that some nations use the days, months, years format, while others type months, days, and years. Allow the opertor to use both methods. Let the user type in spaces between data and instructions. Just ignore them if they serve no purpose.

The operator will make mistakes, and this can petrify the raw beginner. Most mistakes are simple ones. The computer should display a simple message like 'I don't understand that instruction, please try again'. Never flash messages on and off the screen (at least wait for a key to be pressed), nor should you force the operator to read a message (by pausing the program for 20 seconds). You may have displayed a warning on the screen, but the operator who is looking away, or down at a record book, will not see it.

In business, a definite 'no-no' is sound. Soft sounds, that tell the operator a keystroke has registered, are fine (some people still find this irritating). Other soft sounds are acceptable too, but never give a loud warning sound, or make annoying or unnecessary noise. The program you test for three weeks might be used for years. A user's nerves are the first to go. Always give the option of 'turning off' the sound effect. This is particularly important when people close by have to concentrate on something else.

The operator should never be forced to do any 'standard' procedure like sorting. If the operator is only keeping a record to print mailing labels, who cares how they come out? Always give them the option to override a function even if it's standard practice. The operator may simply want to leave quickly, and not wait for the computer to organise its files all at once, late in the day.

Good business programs are considerate ones. Besides letting the operator get away with many non-standard procedures, they do important I/O safety checks. They determine if I/O devices, like disk drives and printers, are functioning before sending information. There is *nothing* worse than a program 'locking up' because data was sent to a peripheral that was not turned on. This is sure to leave the unexperienced operator panicky. Worse still, the user may not even have the peripheral there at the time. A program freeze can destroy the entire record or lose a day's data. Operators have the right to terminate printing of a report etc., at any time. Checking for I/O devices is not difficult. Refer to available technical information regarding the device, or any of your own computer's reference material. Normally a bit is set somewhere in memory (memory mapped), or a port

is sent information (port addressed) to indicate if a peripheral is ready to receive data.

A makeshift timer, made up of a series of counters, can be used to keep track of the last user input. If the operator has not pressed a key, in say, two hours, automatically save the information to disk. This ensures that there is little chance of losing information.

When information is lost, an accidental rebooting of a disk etc., then put recovery routines in the program. These should restore all data lost when the program was reloaded in memory, or better still, recover data that was accidentally deleted. To do this, you must move 'end of data' pointers backwards and forwards in memory rather than actually deleting text. The text should still be there, only out of the file's reach.

Regardless of the mistake, the beginner must never be allowed to ruin the program. It has to be completely 'idiot-proofed' for all conceivable mistakes. All possible bugs have to be removed. No key should be able to destroy the video display image. A bad entry can never be allowed to break up the picture. Try to avoid the scrolling of information. This is only removing 'junk' in a very slow and unprofessional manner. Clear sections of the VDU containing unnecessary information immediately. Try the program out on complete laymen and see how they handle it. Seek opinions and criticisms concerning the basic format and layout of the program.

Carefully consider the market you are aiming at. If the Z80 is cheap and inexpensive then your program should be aimed at the hobbyist. Likewise, an expensive Z80 based machine should be supplied with powerful, but user-friendly software.

Look at what the competition has to offer, and examine the features you have to beat. Go into the local computer store and read all the documentation you can. You might even try getting a demonstration of it from the salesman. If the price is reasonable, you might even consider buying it. Read about them in as many advertisements and magazine reviews as possible. After looking at all the packages, try and determine a few things: Is it even worth writing? With ten or twenty packages on the market, who would be interested in another one? Does it really have an edge on the competition? Does it do something useful and important that the others do not? You should try and base your program on at least one such feature. If it is worth writing, what is the most popular one? Besides adopting many of its ideas, you might have your keyboard layout consistent with it. This will encourage people who have already purchased that product to go over to yours.

Everyone obviously dislikes learning a new program from scratch, especially when you have grown satisfied, or at least comfortable, with the present one. Better still, make your software 'integrated'. In other words, it can load and run files created by other popular programs. A word processor that creates personalized letters from a mailing list database program is 'integrated'. This might seem like a particularly sneaky thing to do, but do not doubt for a moment that the other guy is not going to do the same thing to you. Your only defence when this happens is to look at their extra features and (with the notes you kept on your program) update yours.

Unfortunately, your particular system is unlikely to be complete. Compatability with printers can be the biggest problem. For example, if you own a daisy wheel, it is unlikely your program will handle dot matrix graphics, simply because you have not had the opportunity to use one. Conversely, you might own a dot matrix, and ignore complex justification functions found on daisy wheels. Try and make your program as compatible as possible with as many printers as possible. This requires you to go out and look for the most popular printers.

Printers underline, boldface, and justify in different ways, so many of the better programs allow you to save the 'code sequences' that activates these functions. The program, when requested to underline, for example, will call the buffer, and send the necessary codes to the printer. The printer will then, hopefully, be able to display the information in the format you requested. In effect, it makes your program – with the help of the operator – compatible with all brands.

Should your program conceivably work with just one disk drive, keep the minimum system requirements down to just that. The fewer the specialized hardware requirements, the larger the potential market. An alternative is to release several versions. There might be a 24K or 48K version, or a one or two disk drive version. Most software houses dislike multiple versions however, unless there is a very good reason. A program selling for $80 retail, will probably be available to distributors at $50 or $60. They simply can not afford the cost of stocking many versions of the same product. Versions can be confused, and people can buy the wrong version. Documentation has to compensate for inconsistencies as well.

Try and produce something that is reasonably standardized. Show a software house your product, and if they show interest, they can often supply you with information, or even printers and drives, that will enable you to give it greater compatability. One of the largest markets, and also hardest to compete in is CP/M; a standard DOS for Z80 based computers. Here compatability is essential.

There is a lot of work involved in writing even a simple game. Since the retail price is lower, volume is your concern. First of all, is there much call for games? Are there so many games on the market now, that sales would not be very

remunerative? Alternatively, is the market still fresh, with the chance of making yourself a small fortune? Eventually the only way of really finding out might be writing one (although try and make a few inquiries). On what should you base your game? You might have seen a game in a shop window, or in an arcade centre, and decided to base yours on that. I seriously recommend you forget that idea.

A version of a successful video game is sooner or later brought out for home computers. This could be done legally, by the copyright holders or by someone who has licensed it. Some people just steal the idea of a game, and produce their own version illicitly. So many firms have done this, that it becomes difficult to enforce all copyright laws. If you bring out an illegal version, nothing may happen to you personally, but it could fail or be rejected for a number of other reasons.

You might like a particular firm to market your product. If it's a major distributor with a massive marketing potential (which means big royalties if they take it), they might already have had one or two legal battles over copyrights, and will shun your product because it is not worth their trouble. If it is accepted, or accepted by a smaller distributor, many other distributors may bring out similar versions as well. Not to mention the appearance of the 'official' game.

Resultantly, you are not only in competition with every other game on the market, but five or six programs similar to yours. If your idea is original, it might be unknown, but if genuinely good, distributors will make the effort to promote it sufficiently anyway. Designing a totally original program is more fun. Copying an idea is a gamble that you might lose.

Look at the standard expected from games before you begin yours. Read through magazines to determine if your idea is original, and go over reviews. What do people like about games? If the majority of games available for your computer have mediocre sound effects, spend time in this area. Do any have voice sound? Do the current games give you the top ten or top fifteen high scores? Include that feature when you can.

What about joysticks or the keyboard? Always give the player the choice of both when possible. Let the players choose the keys they want to use, or at least make it consistant with the majority of games on the market. Does the game have the same level of difficulty? Many software reviewers dislike high speed games. Variable speed for beginners or experts will keep more people happy.

If you happen to be the fastest five fingers in your block, and tend to write fast games, remember that many good customers will only play your game occasionally. They may like arcade games in general, but are not very good at them.

We all have a good idea about who plays video games the most, but don't dismiss the rest. Businessmen, parents, grandma – and they are the ones who don't have to save up pocket money to buy your next release. Having read 'fan mail' and other various comments from a software house's customers, I was surprised to find just who played them. So, anything too fast is sure to put them off. You want to build up a reputation, and a clientele; do that by being reasonable.

Why does the player hang on with aching fingers instead of turning the computer off hours earlier? Probably to beat a high score. The fun pales when you can triple your score on a lower level, but get nowhere near that on a higher one. The game gets too easy – after a lot of practice – and it is not worth trying to beat a previous high score on such an easy level. Therefore include difficulty levels, but scale up scores with each harder level. And make sure the points are attractive enough to encourage them to move on to harder levels. Always try and hold the player's interest. If you keep him a long time on one game (whether it is the best he's played or not), he will be sure to buy your next.

What about one or two player options? How many chances do you give a player before the game ends? Always give away bonus 'lives' (extra chances if they get destroyed) at point levels. Try and include outstanding features (easier said than done), like three dimensional graphics, or adventures that communicate in fluent English.

Add variety to games that tend to become monotonous. Put flagships in, or spaceship docking, challenge levels or whole new frames. Move the scenery around a lot too. Why not fight aliens on Earth, in space, and then on Mars? Use different background colours for progressively more difficult stages. Make stars move in the background, but make them move fast! It makes the program seem full of high speed action, even if it's not that difficult.

Give the player a mission for encouragement. The destruction of a mothership or base, the arrival at Pluto or whatever. Games can be fun to play because of their impressive sound and graphics, because they have that irresistible 'addictive' quality, or because it offers a challenge, impossible to refuse. If you can give your game all three qualities, you have a bestseller. Make the challenge hard, but too near to impossible. Make sure *you* can win the game.

If you can afford it, time spent on detail is never wasted. Video games should look flashy. Have explosions, zaps and attractive headings. They will all help put your product ahead of the rest. Once you have written them, use them again in your next game.

8.4 DOCUMENTING YOUR SOFTWARE

Programmers have to write clear and coherent documentation, which is going to be just as much a part of a software package as the program. Only the programmer knows everything about the program; its features and drawbacks, its shortcuts and idiosyncrasies. If you have interested a publisher in your program, sales staff will probably help bring your documentation up to a high standard, but in many cases this is too late. Many publishers will consider not only the quality of the program, but the quality of the documentation before marketing your product. Only documentation can show the publisher how really good your program is.

If the documentation is confusing the program seems confusing, as well as difficult to operate and poorly thought out. In business, good documentation is essential. You could impress the publisher with your software, if you had the documentation to highlight it.

The instruction book is there to teach the user how to use your program and nothing else. An editor/assembler manual is not there to teach Z80, nor a general ledger program to teach business management. However, if your program is possibly aimed at a complete novice, as well as the expert, it would be valid to include appendices with reference tables to help the operator find information for your program – such as a listing of Z80 instructions or other data about the topic in general. Repeat elementary concepts if your program is a 'new concept in business management', but only for reasons of contrast.

Take the view that this is the user's first program. Explain what a 'label' or 'pseudo-op' is because they are editor/assembler features, not necessarily machine language ones. The same rule applies with other software. Complex, (beyond simple experience) economic terms should be defined. The complexity of your program will determine the complexity of your explanation. For a machine language debugging utility it would be safe to say: 'which block moves nnnn to nnnn for nnnn number of bytes' without needing to clarify the word 'block move'. You *do* have to explain what keys to press, and what acceptable parameters exist. Know who you are writing for.

Do not leave information out; do not leave anything vague or ambiguous; make sure all material is double checked for accuracy; do not make the language grandiloquent (like that word) and do not leave any part of the booklet disorganised. Never be tempted to lump unassociated information in one big section. Everything separate should have a separate heading.

It's convenient to put loading instructions, or a summary of them, in the inside cover, since this is the first thing to be read and the first thing to be done. You can

cover loading instructions in more detail further on in the manual. Introduce your reader to your program by stating what it is and what it can do. In particular, why their purchase was such a good one (and a bargain at that). You might start off with an 'Introduction' and then continue with a 'Features' section.

Check carefully that all your words will be completely clear. A novice may not necessarily know what a 'field' is in database management or the exact difference between a 'record' and a 'file'. I expect you to know what I mean when I say, 'the quicksort subroutine uses a recursive algorithm' and even if you have forgotten a word, you could look it up in the glossary section. Make these differences clear right in the beginning, using an analogy where possible.

The hardware requirements, both mandatory and optional, should be covered on the back page of the booklet. The optional features of the program should always be covered last. Once the opening section is completed, clear up all potential misunderstandings and then discuss loading instructions in detail. And cover all aspects of this (every key-stroke) painstakingly. Never expect anything to be 'obvious' such as pressing ENTER or RETURN after typing. All things are alien to a beginner.

Tell them what to expect on the screen as parts of the program load and how long each procedure will take. You might like to insert a 'CAUTION' or a 'WRONG' where there is a probable chance of the user doing something else. Understandably, operators like to be using their program while reading the instruction booklet. A photograph of the screen after loading is ideal (although for the initial draft sent to a publisher a brief explanation of what to expect will do).

Explain the program's operation by running through it with the user. Tell them which keys to press and explain what and why things will happen. Point out how they can correct syntax errors or go back to a previous part of the program. You might even put an error in on purpose, demonstrating how it can be corrected. There are exceptions to the rule, especially when the program is too complex to run through completely. A typical example is a powerful word processor.

While vital to cover important details such as cursor movements and basic editing features like overtyping, inserting and removing text, the rest can be placed in a separate section, each with a separate heading, and a detailed description. Try and cover at least all fundamental principles of operation. The user can be left to hunt additional information – but not completely new concepts.

After your loading instructions and running instruction you will need a table of contents, an index and possibly appendices. Summaries are always handy, and should be used to outline specific operations. For example:

DELETING A RECORD
1. Select CREATE A FILE
2. Answer file query with file to be deleted.
3. Answer delete query by typing *(Y)*
4. Press *(ABORT)* key

Possibly in the form of a summary card that can be removed from the manual.

Always try and pay as much careful attention to the complex concepts in your program as the simple ones. A sophisticated aspect of your program doesn't warrant a sophisticated explanation. You will just find that no one will use the powerful routines you spent so much time developing, and customers will be less impressed with your work.

Some manuals include technical information about the mechanical aspects of the program. Flowcharts, runcharts, important CALL addresses and memory maps. For a computer language, especially a low level language, this would not just be useful but essential. While there are arguments for and against including technical information in business programs, most manuals do avoid the subject. It is a nicety, but *really* not essential.

Working straight out of your head, without at least some sort of skeletal outline, is going to confuse your writing efforts. Inevitably you will cover something too briefly, leave a lot out, and ramble on. Before starting, write points and brief notes (about one sentence) that covers a section of your program. For example, you might note:

– theory of flight
– lift and drag
– motion
– accelerating and stalling
– the engine, fuel
– instruments, the keyboard
– demonstrate flight
– take off/landings

A typical collection of ideas for a flight simulator game. They might eventually grow to several pages in length. In time these ideas will be grouped together under the appropriate chapters and then headings. But no matter how thorough you think your note-taking is, repeatedly run through the program, one slow step at a time, looking for anything you might have left out.

You will have to watch slang and jargon when writing for beginners. A glossary of terms is good, but no excuse for introducing more than essential jargon. Avoid

using the word 'jump' on one page, 'branch' on another and 'goto' on the third. Be consistent. Technical jargon is unacceptable, unless the user is very likely to be familiar with it. So that in: 'The buffer has a maximum limit of 35 kilobytes (K), and of this, 5K is reserved for . . . ' the abbreviated term was shown before it was used. Even the word 'byte' is not necessarily ideal. Rather, the word 'character' could be used. A complete non-technical approach would have been: 'The program can store up to 35,000 characters . . . ' Conversely, if you're writing for an expert, the word 'character' could seem intimidating.

Once you have written the documentation make a carefully thought out and thorough contents page, an index is a better addition. Always try and include an 'in case of trouble . . . ' section, writing down everything you can think of that could go wrong and what can be done, if anything, to correct it. In any program you would have to note all the possibilities of information loss – power failure, bad disk save, etc. Of course, there *should* always be recovery procedures in such programs.

Do not say 'enter the following information', but 'type in the following information'. Do not say: 'Shell sorting' when 'sorting' will do. You are not out to impress anyone, and unfamiliar words are not going to do it anyway! Review carefully what you have written. Have you said what you wanted to say? And does each sentence express the meaning intended? Does one thing lead on logically to another? Good documentation will make your program better.

8.5 SELLING YOUR SOFTWARE

I have emphasised a professional approach when programming, which should, by this stage, read 'commercial' as well. If we were to take a percentage look at the computer programmer population, only a small section, although still significant (competitor-wise), would be versed in machine language. Of these, only a minute fraction would actually write the software we see on the shelves. These are the few who 'got serious' about their hobby, worked hard, and made money.

The odds at first are going to seem formidable. Your routines will not work. Sometimes you will have to give up out of sheer desperation and start again. The job was made no easier for any other programmer, and if you are impressed by any software, think of the trouble that must have been involved in writing it. You may not have to, you might be going through it now. Keep in mind that the really frustrating times (some professionals tend to forget how *really* frustrating they were, and so continue to complain) will pass.

There will always be bugs (first law of the computer programmer's handbook, page 1, paragraph 2), but they will begin to change your routines from ones that 'don't work' to ones that 'don't do what you want'. There's a big difference. The first kind are frustrating and make even the most placid programmer belligerent. The second are a drawback of the trade, which, as some consolation, give you a feeling of conquest once eliminated. Having stuck it out long enough, your bugs become bearable.

There is a lot more to a good program than good programming. There has to be a flow of good ideas, imagination, versatility and power. To do this you have to have confidence in your ability as a programmer, and you can only gain confidence by patient practice and dedication. So, you have written it and have confidence in it. How do you sell it?

There are two ways: marketing it yourself or taking it to a publisher. Before you think of marketing it yourself, consider the trials and tribulations involved. You will have to handle the advertising in magazines. This means organising artwork, typesetting, advertising space, and then paying the artist in advance of any returns. If the advertisement is unsuccessful for any one of a million reasons (wrong magazine, wrong time of year, competition with other firms, too low or too high pricing, etc.) you will have to pay an expensive bill. A lot of folly can go into advertising, so don't get the impression that the 'other guy' is making a fortune. Much of it can be 'image advertising' with products just covering costs.

You will have to purchase mailing lists from direct mail brokers or computer magazines, pay for postage, cost of the tape or disk, the paper for the documentation and wrappings. Those attractive full colour packages you see on the computer shelf cost a fortune. Wouldn't you rather see your product in one of them? If you are ripped off there will be no company to fight for your product's rights. It will take hours upon hours to manually copy the tapes or disks or EPROMS, and there is no way you can compete with a large established publisher. They have the distribution and retail contacts and you don't.

Marketing is a full-time job. If your program does become successful you can bet to lose half your sales, unable to meet demands on your low production capacity.

You can make money out of it, possibly a lot, but not as a hobby.

A publisher takes the risks and responsibilities. A big software company already has a retail network, money, and people dedicated to turning your code into cash – for you and them. We've already looked at much of what publishers want and expect from a good product; standardization, originality, user friendliness, documentation, speed, perhaps colour, graphics, playability, and many other

details depending on the software itself. You will have to find the right publisher however, and expect some rejections.

Some firms produce all their own software, and will not distribute for a freelance. Others will specialize in games or business software and will not be interested in your type of program. Many may merely say your product is 'unsuitable' for their needs. More than once a program has been rejected as 'unsuitable' by a potential software house before being turned into a best seller by a rival firm.

A good way of locating a potential publisher is to search through magazines. Try and approach a company that is selling a lot of software compatible with your own. They will probably have the best contacts. Once you've found a potential firm, ring them up and ask for their marketing or software submissions department (or just explain what you want and ask who to speak to). Talk about your product and see if they have an interest. Point out of course that it's in machine language and that it uses one of the fastest and most sophisticated sorting algorithms known (Quicksort for example) or something to that effect. If they show interest, send them your product.

You might want to request a 'non-disclosure' statement which says something to the effect of: 'I, as a representative of SuperAction Startling Software Surprises, agree to keep the contents of your work strictly confidential.' This stops them distributing 'review' copies to all their friends and the company down the road. While of course only an unethical firm would do such a thing, you don't really know if they're unethical or not until you're bitten. Nevertheless it is a fairly standard procedure and there is no reason why you shouldn't try to protect yourself as much as possible.

Alternatively you can write a letter outlining your product. What it does, how it's done, and what system is required to do it. This can be cheaper, especially if you are making many inquiries from a distance. A large firm may even reply with an international phone call if they see potential. You could be referred to another firm that handles the marketing of products for the distributor. While you obviously can't lie, don't be too modest about your product. If you are writing for a well established market (a computer that has been around for a few years) and approaching a large company, they may receive well over 5000 product inquiries a month.

Many publishers will be willing to take the time to offer advice, provided that they feel your product has potential but is still unsuitable.

If you live some distance away, selling your software is a matter of faith and hope. Not very reassuring words. Software houses have been known to issue first class plane tickets to (probably suspicious) programmers for finalizing a contract, but

they have to be fairly eager to market your software. Such a move is unnecessary, and I am unaware of any firm ever offering an *international* trip.

Copyrights vary from country to country. Some have yet to awaken to the fact that programmers want their work protected, while others allow you to submit program listings, for a reasonable price, for registration. The copyright of a product is always the author's. The publisher agrees to license the program, unless you have sold it outright. Many programs slip through copyright laws, unless the program is so blatently similar to another (such as the same one slightly modified) that an injunction, not allowing it to be sold, is placed on it. Other firms, especially arcade game manufacturers, are very aggressive and will take you to court, if not to win, to try and protect their rights. Ensure your work is completely original. A publisher might be forced *not* to sell your product because you used somebody else's 'idea'.

A publisher, having accepted your product, should go about helping you make as much money out of it as possible. This could mean supplying you with further equipment for conversion over to different systems. Prepare documentation, based on your product for sales and marketing. This includes distribution and advertising.

Although there have been circumstances where a programmer has sold his product outright, a royalty is usually fairer to both parties. A royalty rate in double figures is reasonable, but a few (disreputable perhaps?) have been known to offer less. I would rather file my program away in a cupboard, never to see light again, then yield to an unreasonable royalty, but then, others have gone ahead and made a significant amount of money anyway. A *fair* royalty rate is around 30 per cent after various costs and overheads are subtracted. This comes to around 10 per cent of retail price, on average. Only in exceptional circumstances do software houses offer advances.

After the program is accepted, the programmer can expect to do some adjustments for the final version. This can mean the addition of a particular feature, or just a distribution notice. A successful program can sell in excess of 50,000 copies, but it often is a matter of being in the right market at the right time with the right idea. Don't give up the day job, but soon, just possibly, you may find you no longer need it.

APPENDIX A

GLOSSARY OF TERMS

(Definitions are completely accurate in Z80 machine language only. Slang jargon, although kept to a minimum in this book, is popular in computing magazines and similar material, so appears here in *italics*.)

A

ABSOLUTE ADDRESS/ABSOLUTE ADDRESSING. Where the instruction contains the actual address. E.g. JP 5000H is an absolute address, where JR $ + 54 uses an offset.

ACCUMULATOR. The A register.

ADDRESS. A 16-bit long memory location. All instructions must be assigned an address.

ADDRESSING MODE. The specific way in which an address is represented in an instruction. There are 10 addressing modes: Immediate, e.g. LD A,1; Immediate extended, e.g. LD HL,1; Relative, e.g. JR $ + 50; Extended, e.g. LD A,(5000); Indexed, e.g. LD A,(IX + 1); Register, e.g. LD A,B; Implied, e.g. SUB B; Register Indirect, e.g. LD A,(HL); Bit, e.g. SET 1,A; Modified Page Zero, e.g. RST 0.

ALGORITHM. A 'formula' for solving a problem.

ALPHA-NUMERIC. The letters of the alphabet and the numbers 0-9.

ARCHITECTURE. The organisation and structure of the Microprocessor instruction set.

ARGUMENT. The value(s) entering a subroutine for various processing. The subjects of a subroutine.

ARITHMETIC SHIFT. A shift operation that retains the sign bit.

ARRAY. A 'table' of related data items. See LINEAR ARRAY, MATRIX.

ASCII. American Standard Code for Information Interchange. The most popular character set 'standard'. Codes 0-31 are assigned special VDU display functions analogous to typewriter linefeeds, carriage returns, backspaces, etc. The rest of the 7 bit code represents alpha-numerics and special symbols. E.g. Code 65 represents the letter 'A'.

ASSEMBLER. See EDITOR/ASSEMBLER.

ASSEMBLY LANGUAGE. The mnemonic representation of machine code. Compiled into actual machine code by an editor/assembler program. E.g. 21H,10H,50H of which LD HL,5010H is the mnemonic form. See MNEMONIC, EDITOR/ASSEMBLER.

B

BASE ADDRESS. The starting address (usually of an array).

BAUD. Bits per second. The speed at which information is received or sent by a computer, from or to a peripheral device (serial data).

BCD. Binary Coded Decimal. A special method of working with numbers. They consist of two 4 bit digits holding values between 0-9 in each bit. A byte can contain two BCD numbers, one in each nibble.

BINARY. Base two. E.g. 00010111B.

BINARY CODED DECIMAL. See BCD.

BINARY SEARCH. A searching method that repeatedly subdivides a list in half until left with one element.

BIT. Smallest possible unit of memory, has value of 1 or 0.
— In Z80, to test a bit, see TEST.

BLOCK. A section of memory.

BLOCK COMPARE. A compare that moves sequentially through a block of memory. E.g. CPIR.

BLOCK MOVE. An instruction that moves a block of memory from one address to another.

BOMB-OUT. See *CRASH.*

BOOTSTRAP. The first program on a disk written to load other programs.

BRANCH. To jump. A jump instruction. E.g. JP Z,5000H.

BREAKPOINT. A point at which a program's execution terminates, returning to a monitor program to display the contents of each register at this point. A breakpoint, nothing more than a jump instruction, is inserted and removed by a monitor program. See MONITOR.

BUBBLE SORT. A sorting technique where terms are swapped according to their positions in a list.

BUFFER. An area of memory used to hold data, which, usually, is to be transferred to another area of memory. Any area of memory that stores data only temporarily.

BUG. An error in a program.

BUMP. To increment (rarely decrement).

BYTE. In Z80, an 8 bit unit of memory. E.g. 0FFH.

C

CALL. To execute a subroutine.

CARRY/CARRY FLAG. A bit that is automatically set during an overflow. E.g. test carry; JR C,5000H.

CELL. The smallest 'unit' of a maze, representing a part of a corridor or wall.

CHAIN. A linked-list. See LINKED-LIST.

CIRCULAR SHIFT. To rotate. A rotate instruction. E.g. RRCA.

CLEAR. To load with zero. E.g. LD B,0 clears the B register.
— To load with ASCII 32. E.g. 'Clear the VDU'.

CLOCK. See OVERFLOW.

CLOCK CYCLES. See T STATES.

CLOCK SPEED. See CLOCK FREQUENCY.

CLOCK FREQUENCY. The internal timing used by a microprocessor. measured in MHz. The higher the MHz, the faster the operating speed of the computer.

CODING. Programming a computer. Putting computer instructions together.

COMPILE. To produce machine code from a symbolic high level language.

COMPLEMENT. To Invert. E.g. The number 11011111B complemented is 00100000B. See TWO'S COMPLEMENT.

CONCATENATION. To join/combine two strings together.

CONSTANT. A value used in an operation that does not change (usually throughout the entire execution of the routine). E.g. In ADD A,B the accumulator is the variable, 'B' the constant.

COUNTER. A register that keeps track of the number of times a function must be performed.

CRASH. To fail to function properly. A program that crashes can erase itself from memory or do some other damage to the code.

CP/M. Control Program/Microcomputer. A 'standard' disk operating system for Z80 computers, developed by Digital Research.

CPU. Central Processing Unit. A part of a microprocessor.

D

DEBOUNCE. A delay loop used to avoid detecting invalid information. E.g. A keyboard scanning routine has a debounce delay to avoid keys being registered more than once during a press.

DEBUGGER. See MONITOR.

DEBUGGING. To correct programming errors.

DEFAULT. The value that is used when no value is specified by the operator.

DIRECT MEMORY ACCESS. The direct transfer of peripheral data to memory (without CPU control).

DISK OPERATING SYSTEM. See DOS.

DIRECT ADDRESSING. See ADDRESSING MODE.

DISPLACEMENT. An offset. E.g. In the instruction LD A,(IX + 50) the base address is in IX, the displacement is 50.

DOS. Disk Operating System. Collection of system programs that manage disk space allocation, and the loading and saving of software, etc.

DUMP. To move an area of memory to an output device. E.g. Printer, disk drive, cassette recorder, etc.

DUMMY. Something that is there to fill a parameter, but is itself of no consequence (not used).

E

EDITOR. See EDITOR/ASSEMBLER.

EDITOR/ASSEMBLER. A program that allows you to type in mnemonics – assembly language – which can be edited (the editor) and then assembled (the assembler) into actual machine code. See ASSEMBLY LANGUAGE, MACHINE LANGUAGE.

ELEMENT. A term in linear array. See TERM, LINEAR ARRAY.

END. An editor/assembler pseudo-operation. Instructs the assembler to terminate assembly.

ENDLESS LOOP. Any instruction that jumps to the beginning of itself. E.g. JR $.

EQU. An editor/assembler special label which assigns a value to a name. E.g. If 'COUNT EQU 5000H' COUNT can be used in place of 5000H.

EQUATE. See EQU.

EXCLUSIVE OR. To XOR.

HIGH LEVEL LANGUAGE. A compiled or interpreted language in which there is no need to work directly with Z80 registers and conditions. High level languages are designed to make programming easier.

HOUSE KEEPING. The 'extra' work done by a routine before the main task can be performed.

I

IMMEDIATE ADDRESSING. See ADDRESSING MODE.

IMMEDIATE EXTENDED. See ADDRESSING MODE.

IMPLIED. An assembly language instruction where two registers are manipulated, but only one is represented in the mnemonic. E.g. SUB B. It is implied that the accumulator is the other register. E.g. DJNZ $. It is implied that B is to be decremented.

INCLUSIVE OR. An 'OR' instruction.

INDEX. A table holding addresses pointing to the locations of data.

INDEXED ADDRESSING. See ADDRESSING MODE.

INDEX REGISTER. IX or IY.

INDIRECT ADDRESSING. See ADDRESSING MODE.

INPUT/OUTPUT. The sending or receiving of information. E.g. VDU, printer, keyboard, disk drive, etc.

INSTRUCTION. A computer operation.

INSTRUCTION LENGTH. The number of bytes needed to define an instruction.

INSTRUCTION SET. All the available machine language instructions. E.g. DI, INC A, BIT 3,C etc.

INTERPRETIVE EXECUTION. To execute a program, while still under control of the executer. See SINGLE-STEPPER.

INTERRUPT. A signal that stops a computer executing one program or routine so that it can execute another program or routines. Used to 'simulate' performing two tasks at once.

INTERRUPT-DRIVEN I/O. The use of interrupt routines to read and write data to peripherals.

INVERT. See COMPLEMENT.

ITERATION. Repetition of an action or process. E.g. A loop that is executed five times has five ITERATIONS. In a program, each time PC returns to an address it previously equalled.

I/O. Input/Output.

I/O MAPPED I/O. Port addressed I/O.

J

JUMP. A jump instruction. E.g. JP 5000H.

JUMP TABLE. A table that holds jump addresses. Possibly an index. See INDEX.

L

LABEL. A name given to an address for use in an editor/assembler.

LEAST SIGNIFICANT BITS. See LSB.

LEAST SIGNIFICANT DIGIT. See LSD.

LINEAR ARRAY. A collection of terms; bytes, words or strings, that follow one after the other in sequential order through memory.

LINKED-LIST. A list of blocks of data, with a pointer in each block pointing to the location of the next block.

LOCK-UP. Computer that does not respond to user input, nor does anything constructive. Usually as the result of a *crash*. See *CRASH*.

LOGIC. Usually; machine language instructions. The instructions that make up a routine. The instructions used to perform the algorithm.

LOGICAL OPERATION. Instructions that AND, OR or XOR data.

LOGICAL SHIFT. A shift operation that moves a bit, normally zero, into bit 7 or 0 according to the direction of the shift.

LOGIC FAULT. An error in a program's algorithm.

LOOP. Set of instructions that are executed repeatedly.

LSB. Least Significant Bits. Bits 0-3. The right nibble.
— Rarely: just bit zero (but not in this book).

LSD. Least Significant Digit. The right nibble of a hexadecimal number. E.g. The LSD of 67H is '7'.

M

MACHINE CODE. See MACHINE LANGUAGE.

MACHINE LANGUAGE. The internal, numeric, language of a computer.

MACHINE CYCLE. Three or more T states. E.g. DAA takes one M cycle.

MACRO. A predefined set of instructions. Used in an editor/assembler.

MASK. To clear specific bits from a byte. E.g. AND 50H.

MATRIX. A table. Data that is divided into rows and columns.

MEDIA. The material that non-addressable information is stored on. E.g. disks, magnetic tape, etc.

MEMORY MAP. The specific organisation of a computer's memory into ROM, RAM and I/O.

MEMORY-MAPPED I/O. I/O ports that are assigned memory addresses. E.g. Memory-mapped computers have addresses that change according to the press of a key.

Mhz. See CLOCK FREQUENCY.

MICROCOMPUTER. A computer that has a microprocessor.

MICROPROCESSOR. The data processing unit of a computer. E.g. The Z80.

MNEMONIC. The symbolic representation – assembly language – of machine code. E.g. 76H is represented in mnemonic form as 'HALT'.

MODIFIED PAGE ZERO. See ADDRESSING MODE.

MODULAR PROGRAMMING. A programming technique that divides a program into separate 'modules' of independant logic. This divides all code into subroutines.

MODULE. A portion of a program. A subroutine.

MONITOR. A program used as a programming aid. Helps the programmer to do some or all of the following: debug, modify, run, single-step, disassemble, repair or write a program.

MOST SIGNIFICANT BITS. See MSB.

MOST SIGNIFICANT DIGIT. See MSD.

MPU. Microprocessing unit. The microprocessor.

MSB. Most Significant Bits. Bits 7-4. The left nibble.
— Rarely, just the 7th bit (but not in this book).

MSD. Most Significant Digit. The left digit of a hexadecimal number. E.g. the MSB of 56H is '5'.

N

NEGATE. To subtract from zero. To complement and add one. A Two's complement.

NIBBLE. Half a byte. A 4-bit unit. The left nibble of a byte includes bits 7-4, the right nibble bits 3-0.

NOP. No OPeration.

NUMBER CRUNCHING. Doing arithmetic for the sake of arithmetic. Routines that perform calculations like addition, subtraction and division.

O

OBJECT CODE. The machine language output of an editor/assembler, as opposed to source code. A machine language program is the 'object code'.

OFFSET. See DISPLACEMENT.

ONE'S COMPLEMENT. To invert. See COMPLEMENT.

ON-LINE MEMORY. Accessable memory.

OP CODE. An operation code. The part of an instruction that defines the action to perform. E.g. LD, INC, AND.

OPERAND. The subject of an operation. E.g. In ADD A,B the operands are A and B.

OPERATION CODE. See OP CODE.

ORG. Editor/assembler pseudo-operation standing for 'ORiGin' and used to define the starting location of machine language object code.

OS. The Operating System. The programs stored in ROM memory that perform all the necessary functions of I/O control, etc.

OVERFLOW. To exceed the limits of a register by adding two values that cannot possibly be contained in a single register.
— A special flag condition used to check the overflow of signed numbers.

P

PACKED DECIMAL. BCD. See BCD.

PAGE. A block of memory, usually 256 bytes in length.

PAGE ADDRESSING. A method of addressing that makes reference to particular 'pages'. The instruction RST 0H makes reference to page 0. Rarely used. See PAGE.

PARAMETER. A value required by a subroutine/or instruction on execution. E.g. A block move instruction requires source, destination and bytes to move parameters.

PARITY. An error detecting code.

PATCH. To modify a machine language program.

PERIPHERAL. Computer equipment that is not the CPU. E.g. Disk drives, printers, etc.

PIXEL. PIcture ELement. The smallest displayable 'points' on a VDU. Usually represented internally as bits.

POINTER. A 16-bit number that marks where data is located.

POP. To remove from the stack. E.g. In Z80, 'POP HL' recovers the contents of HL from the stack.

PORT. A special address that can be used to send or receive information. E.g. In Z80, OUT A,(PORT) and IN A,(PORT) are I/O instructions.

PORT ADDRESSING. I/O addressing ports. See PORT.

PROGRAM COUNTER. The PC register. The register that keeps track of the next instruction to be executed.

PROGRAMMER'S TOOLKIT. The collection of programs, subroutines, and modules that make up a programmer's reference material. Usually 'tried and true' machine code programs.

PSEUDO OP. An editor/assembler instruction, although not in itself a machine language instruction. E.g. DM, DW, DB, etc.

PSEUDO OPERATION. See PSEUDO OP.

PULL. To POP. See POP.

PUSH. Save a register on the stack. E.g. In Z80, 'PUSH HL' saves HL on the stack.

Q

QUICKSORT. A sophisticated recursive sorting algorithm.

R

RAM. Random-Access-Memory. Memory that can be loaded with values.

RANDOM-ACCESS-MEMORY. See RAM.

READ-ONLY-MEMORY. See ROM.

REAL-TIME. Programs that act at a speed similar to those of the 'real world'.

RECURSIVE PROGRAMMING. Programming technique in which subroutines CALL themselves.

REFRESH REGISTER. Register R, used as a counter to maintain RAM chips.

REGISTER. A special storage location inside a CPU. Data can be manipulated in this location. E.g. In Z80 A, F, B, C, D, E, H, L, PC, SP, IX, IY, A', F', B', C', D', E', H', L', I and R, are all registers.

REGISTER ADDRESSING. See ADDRESSING MODE.

REGISTER PAIR. Registers that can be accessed in pairs. E.g. HL, BC, DE, HL', BC', DE' are all register pairs.

RELATIVE ADDRESS. An address calculated by adding or subtracting a value from the current location. E.g. JR $+12.

RELATIVE ADDRESSING. See ADDRESSING MODE

RELATIVE OFFSET. The displacement in a JR instruction. E.g. In JR $+45 the '45' is the displacement. See DISPLACEMENT.

RELOCATE/RELOCATABLE. To move to another area of memory. Machine code that can be moved anywhere in memory and function the same way without changes.

RETURN. Exit from a subroutine. E.g. In Z80, 'RET'.

ROM. Read-Only-Memory. Memory that cannot be loaded with values. Its contents remain constant.

ROM CALL. To execute a subroutine in ROM memory.

ROM INDEPENDENT. Makes no ROM CALLs. A program that functions regardless of the computer's ROM.

ROTATE. A bit shift that moves bit 0 or 7 into 7 or 0 according to the direction of a shift. No bits are lost during a rotate. E.g. RRCA. See WRAP AROUND.

RUNCHART. Diagramatic representation of the order of execution of a program.

S

SCROLL. To shift up or down.

SHELL SORT. An advanced bubble sorting algorithm.

SHIFT. To move the bits of a byte one position to the left or right.

SIGN. The positive or negative indicator of a number.

SIGNED BIT. Bit 7.

SIGNED NUMBER. A number which reserved the 7th bit to stand for a 'sign'; 1 negative, 0 positive.

SINGLE-STEPPER. A program, usually found in a monitor, that interpretively executes machine language. A program that executes one instruction at a time of another program, so that register contents can be monitored by the programmer.

SOFTWARE. Computer programs.

SOURCE CODE. The mnemonic or high level code from which object code is derived. In Z80 programming, source code is assembly language.

SUBSCRIPT. Variable used to point to an array element.

STACK. A linear array used for storing information temporarily in Z80. A data structure where only the 'top' item can be removed, and where the last item added becomes the 'top'.

STACK POINTER. The SP register.

STATES. See T STATES.

STRING. A linear array of characters related to one another. E.g. DB 'THIS IS A STRING'.

STRUCTURAL ERROR. See LOGIC FAULT.

SUBROUTINE. A routine that can be accessed from more than one location in memory.

SUBROUTINE PACKAGE. See SYSTEM SUBROUTINES.

SYMBOLIC. Represented in mnemonic, as opposed to numeric, form.

SYSTEM PROGRAM. Loosely, any machine language program.

SYSTEMS PROGRAM. 'Book keeping' software. Organises memory allocation, file mangement, etc., usually in an OS.

SYSTEM SUBROUTINES. Lowest level module. Subroutines CALLed by more than one module.

T

TERM. An element (byte, word or string) in a list.

TERMINATOR. A data item that indicates the end of a list, array, string, etc.

TEST. To determine its value, usually for bits. E.g. BIT 5,A tests the 5th bit of the accumulator.

TEXT. Words and sentences. Anything in ASCII.

TOKEN. A number used to represent a collection of symbols or a string, in memory.

TRACER. A single-stepper. See SINGLE-STEPPER.

T STATES. The number of clock cycles required to execute an instruction. Used to determine the speed of each instruction.

TWO'S COMPLEMENT. See NEGATE.

U

UTILITY. A program designed to help the programmer. It may be a monitor. See MONITOR.

V

VARIABLE. Anything (address, register) that changes its value.

VDU. Video Display Unit.

VECTOR. Loosely; any branch. In particular interrupt jumps.

VOLATILE MEMORY. Usually RAM. Memory that loses its contents without power.

W

WORD. A 16-bit number. E.g. 5000H.

WORD-LENGTH. 16-bits in length. See WORD.

WRAPAROUND. Data or information that acts as if it were joined at the ends. On a VDU data that exceeds the column length, 'wraps around' to a lower column.

APPENDIX B

PROGRAM CONVERSION

Transferring a Z80 program over to a different microcomputer is in most cases not difficult. Here are the main points to consider:

— Colour
— Graphics resolution
— Addressing VDU memory
— VDU column and row length
— Addressing the keyboard
— I/O (disk, tape, printer, the bootstrap, operating system load and save utilities)
— Available memory
— Compatibility (with disk operating systems, other programs)
— Computer operating speed
— Interrupt circuitry
— Others (bank switching, special addressing techniques)

The importance of each point will depend on how different it is from your current microcomputer. You will have difficulty converting a business program that used an 80 column display over to a 50 column one. In fact you may have to redesign user input. Some keys may not even exist on a different microcomputer. The problem of graphics speaks for itself. Graphics handling in most cases has to be completely rewritten.

Programs using interrupts may not be able to operate on Z80 microcomputers lacking the necessary hardware. The memory limit will determine if it is even possible to write a program over. As well, the program will probably have to be relocated into a more suitable area of memory. The previous area may be ROM in the new microcomputer, or used by an operating system. Memory mapped I/O and port addressing is dealt with in chapter one of this book and is the major area of concern regarding a new microcomputer.

If your microcomputer is a popular brand you will probably be able to find printed information regarding its technical specifications. If your computer has only recently been released on the market, or has a limited distribution, then you are in trouble, although it is probably not totally insurmountable.

After looking through all the book stores and computer outlets without luck, try approaching the manufacturer or distributor for your area. If this returns a blank,

try and contact software houses producing machine language programs for your computer. I've always replied to such enquiries (although these are usually 'how do I generate random numbers' etc., etc., etc.) if the writer is polite enough to include a stamped, self-addressed envelope, although I can't guarantee that others will do the same. Your last choice is to write a disassembler in a high level language and do your own examination of ROM memory. Unfortunately some computers (but not the majority) are so complex that it makes the attempt, without extreme perseverance, nearly impossible.

The following is the technical specifications of three popular Z80 computers:

TRS-80 MODEL I/II/IV (GENIE, SYSTEM 80, TRZ-80)

KEYBOARD ADDRESSING: Memory mapped. Use 3801H (@-G), 3802H (H-O), 3804H (P-W), 3808H (X-Z), 3810H (0-7), 3820H (8-/), 3840H (ENTER, CLEAR, BREAK, up arrow, down arrow, left arrow, right arrow, SPACE BAR) and bit 0 of 3880H for SHIFT. Bits are set when keys are pressed.

VDU ADDRESSING: Memory mapped. Occupies locations 3C00H-3FFFH.

CHARACTER SET: Standard ASCII.

ROM CALLS: 49H is a keyboard scan routine. Character returned in the accumulator. 33AH is a display character routine. Character must be in the accumulator on entry.

SPECIAL FEATURES: None.

ZX SPECTRUM

KEYBOARD ADDRESSING: Addressed to port 254.

VDU ADDRESSING: Special addressing techniques.

CHARACTER SET: Standard ASCII.

ROM CALLS: 2BFH is a keyboard scan routine. Character returned in the accumulator. 9F4H is a character display routine. The character must be in the accumulator on entry.

SPECIAL FEATURES: Not noted here. Refer to SPECTRUM MACHINE CODE REFERENCE GUIDE and SPECTRUM MACHINE CODE MADE EASY – Interface Publications.

LASER 200 (VZ-200)

KEYBOARD ADDRESSING: Memory mapped. Use 6FF7H (1-5), 6FDFH (minus sign, 0, 9, 8, 7, 6), 6FFEH (Q, W, E, R, T), 6FBFH (RETN, P, O, I, U, Y), 6FFDH (CTRL, A, S, D, F, G), 6F7FH (':', ';', L, J, H), 6FFBH (shift, Z, X, C, V, B), 6FEFH (SPACE, ' ', ' ', M, N). Bits are reset when keys are pressed.

VDU ADDRESSING: Memory mapped. Occupies locations 7000H-71FFH.

CHARACTER SET: Non-standard ASCII. Lowercase replaced by inverse. Alphabet begins at character code zero, otherwise standard.

ROM CALLS: 49H is a keyboard scan routine. Character returned in the accumulator. 33H is a display character routine. Character must be in the accumulator on entry.

SPECIAL FEATURES: Computer must be in MODE(1) to activate high resolution graphics.

APPENDIX C

Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	130	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

HEXADECIMAL TO DECIMAL CONVERSION

HEXADECIMAL TO
BINARY CONVERSION

0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

APPENDIX D

ASCII CONTROL CODES AND CHARACTERS

Dec	Hex	Char	Dec	Hex	Char
0	00	NULL	16	10	DLE
1	01	SOH	17	11	DC1
2	02	STX	18	12	DC2
3	03	ETX	19	13	DC3
4	04	EOT	20	14	DC4
5	05	ENQ	21	15	NAK
6	06	ACK	22	16	SYN
7	07	BEL	23	17	ETB
8	08	BKSP	24	18	CAN
9	09	HT	25	19	EM
10	0A	LF	26	1A	SUB
11	0B	VT	27	1B	ESC
12	0C	FF	28	1C	FS
13	0D	CR	29	1D	GS
14	0E	SO	30	1E	RS
15	0F	SI	31	1F	US

Dec	Hex	Char	Dec	Hex	Char
32	20	SPACE	53	35	5
33	21	!	54	36	6
34	22	"	55	37	7
35	23	#	56	38	8
36	24	$	57	39	9
37	25	%	58	3A	:
38	26	&	59	3B	;
39	27	'	60	3C	<
40	28	(	61	3D	=
41	29	)	62	3E	>
42	2A	*	63	3F	?
43	2B	+	64	40	@
44	2C	,	65	41	A
45	2D	-	66	42	B
46	2E	.	67	43	C
47	2F	/	68	44	D
48	30	0	69	45	E
49	31	1	70	46	F
50	32	2	71	47	G
51	33	3	72	48	H
52	34	4	73	49	I

Dec	Hex	Char
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^

Dec	Hex	Char
95	5F	_
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
108	6B	k
109	6C	l
110	6D	m
111	6E	n
112	6F	o
113	70	p
114	71	q
115	72	r
116	73	s

Dec	Hex	Char	Dec	Hex	Char
117	74	t	123	7A	z
118	75	u	124	7B	{
119	76	v	125	7C	|
120	77	w	126	7D	}
121	78	x	127	7E	~
122	79	y	128	7F	DEL

SYMBOLS:

NUL – Null

SOH – Start of Heading

STX – Start of Text

ETX – End of Text

EOT – End of Transmission

ENQ – Enquiry

ACK – Acknowledge

BEL – Bell

BKSP – Backspace

HT – Horizontal Tabulation

LF – Line Feed

VT – Vertical Tabulation

FF – Form Feed

SYMBOLS:

CR	–	Carriage Return
SO	–	Shift Out
SI	–	Shift In
DLE	–	Data Link Escape
DC	–	Device Control
NAK	–	Negative Acknowledge
SYN	–	Syncronous Idle
ETB	–	End of Transmission Block
CAN	–	Cancel
EM	–	End of Medium
SUB	–	Substitute
ESC	–	Escape
FS	–	File Separator
GS	–	Group Separator
RS	–	Record Separator
US	–	Unit Separator
SPACE	–	Blank Space
DEL	–	Delete

APPENDIX E

SUMMARY OF Z80 INSTRUCTION FUNCTIONS

```
ADC   A,operand1    ;Add operand and carry to accumulator

ADC   HL,regpair    ;Add register-pair and carry to accumulator

ADD   A,operand1    ;Add operand to accumulator

ADD   HL,regpair    ;Add register-pair to accumulator

ADD   IX,regpair    ;Add register-pair to IX

ADD   IX,IX         ;Add IX to IX

ADD   IY,regpair    ;Add register-pair to IY

ADD   IY,IY         ;Add IY to IY

AND   operand1      ;Logically AND operand to accumulator

BIT   b,operand     ;Test bit b of operand

CALL  addr          ;Call address

CALL  cc,addr       ;Call address if cc is valid

CCF                 ;Complement carry flag

CP    operand1      ;Compare operand to accumulator

CPD                 ;Compare (HL) to accumulator, decrement HL
                    ;And BC

CPDR                ;Compare (HL) to accumulator, decrement HL
                    ;And BC, repeat until BC = 0 or match found

CPI                 ;Compare (HL) to accumulator, increment HL
                    ;And decrement BC
```

```
CPIR            ;Compare (HL) to accumulator, increment HL
                ;And decrement BC, repeat until BC = 0 or
                ;Match found

CPL             ;Complement (invert) accumulator

DAA             ;Decimal adjust accumulator

DEC   operand   ;Decrement operand

DEC   IX        ;Decrement IX

DEC   IY        ;Decrement IY

DEC   SP        ;Decrement stack pointer

DI              ;Disable interrupts

DJNZ  dis       ;Decrement B, jump to displacement if B = 0

EI              ;Enable interrupts

EX    (SP),HL   ;Exchange (SP) and HL

EX    (SP),IX   ;Exchange (SP) and IX

EX    (SP),IY   ;Exchange (SP) and IY

EX    AF,AF'    ;Exchange AF with alternate AF

EX    DE,HL     ;Exchange DE and HL

EXX             ;Exchange general-purpose registers with
                ;Alternates

HALT            ;Suspect CPU operation

IM    mode      ;Activate interrupt mode

IN    A,(data)  ;Input into accumulator port data

IN    reg,(C)   ;Input into register port C

INC   operand   ;Increment operand
```

```
INC     regpair     ;Increment register-pair

INC     IX          ;Increment IX

INC     IY          ;Increment IY

IND                 ;Load (HL) with input from port C,
                    ;Decrement HL and B

INDR                ;Load (HL) with input from port C,
                    ;Decrement HL and B, repeat until B = 0

INI                 ;Load (HL) with input from port C,
                    ;Decrement B and increment HL

INIR                ;Load (HL) with input from port C,
                    ;Decrement B and increment HL, repeat until
                    ;B = 0

JP      (HL)        ;Jump to address of HL

JP      (IX)        ;Jump to address of IX

JP      (IY)        ;Jump to address of IY

JP      addr        ;Jump address

JP      cc,address  ;Load program counter with address
                    ;If cc is valid

JR      dis         ;Jump displacement

JR      c,dis       ;Jump displacement if c valid

LD      A,I         ;Load interrupt vector into accumulator

LD      A,operand1  ;Load operand into accumulator

LD      A,R         ;Load refresh into accumulator

LD      (BC),A      ;Load accumulator into (BC)

LD      (DE),A      ;Load accumulator into (DE)
```

```
LD      (HL),data       ;Load data into (HL)

LD      HL,(addr)       ;Load (address) into HL

LD      (HL),reg        ;Load register into (HL)

LD      I,A             ;Load accumulator into interrupt vector

LD      IX,addr         ;Load address into IX

LD      IX,(addr)       ;Load (address) into IX

LD      (IX + dis),data ;Load into IX + displacement data

LD      IY,addr         ;Load address into IY

LD      IY,(addr)       ;Add (address) into IY

LD      (IY + dis),data ;Load into IY + displacement data

LD      (addr),A        ;Load accumulator into (address)

LD      (addr),regpair  ;Load register-pair into (address)

LD      (addr),IX       ;Load IX into (address)

LD      (addr),IY       ;Load IY into (address)

LD      regpair,addr    ;Load address into register-pair

LD      R,A             ;Load accumulator into refresh

LD      reg,operand1    ;Load operand into register

LD      SP,HL           ;Load HL into stack pointer

LD      SP,IX           ;Load IX into stack pointer

LD      SP,IY           ;Load IY into stack pointer

LDD                     ;Load (HL) into (DE), decrement DE, BC and HL

LDDR                    ;Load (HL) into (DE), decrement DE, BC and HL,
                        ;Repeat until BC = 0
```

```
LDI             ;Load (HL) into (DE), increment DE and HL,
                ;Decrement BC

LDIR            ;Load (HL) into (DE), increment DE and HL,
                ;Decrement BC

NEG             ;Negate accumulator

NOP             ;No operation

OR    operand1  ;Logically OR operand to accumulator

OTDR            ;Load (HL) into port C, decrement B and HL,
                ;Repeat until B = 0

OTIR            ;Load (HL) into port C, increment HL,
                ;Decrement B, repeat until B = 0

OUT   (C),reg   ;Load register into port C

OUT   (data),A  ;Load accumulator into port data

OUTD            ;Load (HL) into port C, decrement HL and B

OUTI            ;Load (HL) into port C, increment HL and
                ;Decrement B

POP   regpair   ;Recover register-pair from stack

POP   IX        ;Recover IX from stack

POP   IY        ;Recover IY from stack

PUSH  regpair   ;Put register-pair on stack

PUSH
IX              ;Put IX on stack

PUSH
IY              ;Put IY on stack

RES   b,operand ;Reset bit b of operand

RET             ;Return
```

```
RET     cc          ;Return if condition valid

RETI                ;Return from interrupt

RETN                ;Return from non maskable interrupt

RL      operand     ;Rotate operand left through carry

RLA                 ;Rotate accumulator left through carry

RLC     operand     ;Rotate operand left circular

RLCA                ;Rotate accumulator left circular

RLD                 ;Rotate digit left and right between (HL)
                    ;And accumulator

RR      operand     ;Rotate operand right through carry

RRA                 ;Rotate accumulator right through carry

RRC     operand     ;Rotate operand right circular

RRCA                ;Rotate accumulator right circular

RRD                 ;Rotate digit right and left between (HL)
                    ;And accumulator

RST     vector      ;Restart at location vector

SBC     A,operand   ;Subtract operand from accumulator with carry

SBC     HL,repair   ;Subtract register-pair from HL with carry

SCF                 ;Set carry flag

SET     b,operand   ;Set bit b of operand

SLA     operand     ;Shift operand left arithmetic

SRA     operand     ;Shift operand right arithmetic

SRL     operand     ;Shift operand right logical
```

SUB	operand	;Subtract operand from accumulator
XOR	operand	;Exclusive OR operand to accumulator

SYMBOLS

reg	A, B, C, D, E, H, L.
regpair	HL, BC, DE, SP
mode	1, 2, 3
operand1	A, B, C, D, E, H, L, (HL), (IX + dis), (IY + dis), data
operand	A, B, C, D, E, H, L, (HL), (IX + dis), (IY + dis)
dis	displacement offset, a number between 0-255
data	number between 0-255
addr	address between 0-65535
cc	conditions Z, NZ, C, NC, P, M, PE, PO
c	conditions Z, NZ, C, NC
vector	addresses 0, 8, 16, 24, 32, 40, 48 and 56

APPENDIX F

ALPHABETICAL LIST OF Z80 INSTRUCTIONS

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
ADC A,(HL)	8E	C Z P/V S	2
ADC A,(IX+Ø)	DD8EØØ	C Z P/V S	5
ADC A,(IY+Ø)	FD8EØØ	C Z P/V S	5
ADC A,Ø	CEØØ	C Z P/V S	1
ADC A,A	8F	C Z P/V S	1
ADC A,B	88	C Z P/V S	1
ADC A,C	89	C Z P/V S	1
ADC A,D	8A	C Z P/V S	1
ADC A,E	8B	C Z P/V S	1
ADC A,H	8C	C Z P/V S	1
ADC HL,BC	ED4A	C Z P/V S	4
ADC HL,DE	ED5A	C Z P/V S	4
ADC HL,HL	ED6A	C Z P/V S	4
ADC HL,SP	ED7A	C Z P/V S	4
ADD A,(HL)	86	C Z P/V S	2
ADD A,(IX+Ø)	DD86ØØ	C Z P/V S	5

MNEMONIC		OBJECT CODE	FLAGS	M-CYCLES
ADD	A,(IY+Ø)	FD86ØØ	C Z P/V S	5
ADD	A,Ø	C6ØØ	C Z P/V S	2
ADD	A,A	87	C Z P/V S	1
ADD	A,B	8Ø	C Z P/V S	1
ADD	A,C	81	C Z P/V S	1
ADD	A,D	82	C Z P/V S	1
ADD	A,E	83	C Z P/V S	1
ADD	A,H	84	C Z P/V S	1
ADD	HL,BC	Ø9	C	3
ADD	HL,DE	19	C	3
ADD	HL,HL	29	C	3
ADD	HL,SP	39	C	3
ADD	IX,BC	DDØ9	C	4
ADD	IX,DE	DD19	C	4
ADD	IX,IX	DD29	C	4
ADD	IX,SP	DD39	C	4
ADD	IY,BC	FDØ9	C	4
ADD	IY,DE	FD19	C	4
ADD	IY,IY	FD29	C	4
ADD	IY,SP	FD39	C	4
AND	(HL)	A6	Z P/V S	2

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
AND (IX+0)	DDA600	Z P/V S	5
AND (IY+0)	FDA600	Z P/V S	5
AND 0	E600	Z P/V S	2
AND A	A7	Z P/V S	1
AND B	A0	Z P/V S	1
AND C	A1	Z P/V S	1
AND D	A2	Z P/V S	1
AND E	A3	Z P/V S	1
AND H	A4	Z P/V S	1
AND L	A5	Z P/V S	1
BIT 0,(HL)	CB46	Z	3
BIT 0,(IX+0)	DDCB0046	Z	5
BIT 0,(IY+0)	FDCB0046	Z	5
BIT 0,A	CB47	Z	2
BIT 0,B	CB40	Z	2
BIT 0,C	CB41	Z	2
BIT 0,D	CB42	Z	2
BIT 0,E	CB43	Z	2
BIT 0,H	CB44	Z	2
BIT 0,L	CB45	Z	2
BIT 1,(HL)	CB4E	Z	3

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
BIT 1,(IX+Ø)	DDCBØØ4E	Z	5
BIT 1,(IY+Ø)	FDCBØØ4E	Z	5
BIT 1,A	CB4F	Z	2
BIT 1,B	CB48	Z	2
BIT 1,C	CB49	Z	2
BIT 1,D	CB4A	Z	2
BIT 1,E	CB4B	Z	2
BIT 1,H	CB4C	Z	2
BIT 1,L	CB4D	Z	2
BIT 2,(HL)	CB56	Z	3
BIT 2,(IX+Ø)	DDCBØØ56	Z	5
BIT 2,(IY+Ø)	FDCBØØ56	Z	5
BIT 2,A	CB57	Z	2
BIT 2,B	CB50	Z	2
BIT 2,C	CB51	Z	2
BIT 2,D	CB52	Z	2
BIT 2,E	CB53	Z	2
BIT 2,H	CB54	Z	2
BIT 2,L	CB55	Z	2
BIT 3,(HL)	CB5E	Z	3
BIT 3,(IX+Ø)	DDCBØØ5E	Z	5

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
BIT 3,(IY+Ø)	FDCBØØ5E	Z	5
BIT 3,A	CB5F	Z	2
BIT 3,B	CB58	Z	2
BIT 3,C	CB59	Z	2
BIT 3,D	CB5A	Z	2
BIT 3,E	CB5B	Z	2
BIT 3,H	CB5C	Z	2
BIT 3,L	CB5D	Z	2
BIT 4,(HL)	CB66	Z	3
BIT 4,(IX+Ø)	DDCBØØ66	Z	5
BIT 4,A	CB67	Z	2
BIT 4,B	CB60	Z	2
BIT 4,C	CB61	Z	2
BIT 4,D	CB62	Z	2
BIT 4,E	CB63	Z	2
BIT 4,H	CB64	Z	2
BIT 4,L	CB65	Z	2
BIT 5,(HL)	CB6E	Z	3
BIT 5,(IX+Ø)	DDCBØØ6E	Z	5
BIT 5,(IY+Ø)	FDCBØØ6E	Z	5
BIT 5,A	CB6F	Z	2

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
BIT 5,B	CB68	Z	2
BIT 5,C	CB69	Z	2
BIT 5,D	CB6A	Z	2
BIT 5,E	CB6B	Z	2
BIT 5,H	CB6C	Z	2
BIT 5,L	CB6D	Z	2
BIT 6,(HL)	CB76	Z	3
BIT 6,(IX+Ø)	DDCBØØ76	Z	5
BIT 6,(IY+Ø)	DDCBØØ76	Z	5
BIT 6,A	CB77	Z	2
BIT 6,B	CB70	Z	2
BIT 6,D	CB72	Z	2
BIT 6,E	CB73	Z	2
BIT 6,H	CB74	Z	2
BIT 6,L	CB75	Z	2
BIT 7,(HL)	CB7E	Z	3
BIT 7,(IX+Ø)	DDCBØØ7E	Z	5
BIT 7,(IY+Ø)	FDCBØØ7E	Z	5
BIT 7,A	CB7F	Z	2
BIT 7,B	CB78	Z	2
BIT 7,C	CB79	Z	2

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
BIT 7,D	CB7A	Z	2
BIT 7,E	CB7B	Z	2
BIT 7,H	CB7C	Z	2
BIT 7,L	CB7D	Z	2
CALL Ø	CDØØØØ		1
CALL C,Ø	DCØØØØ		3 or 5
CALL M,Ø	FCØØØØ		3 or 5
CALL NC,Ø	D4ØØØØ		3 or 5
CALL NZ,Ø	C4ØØØØ		3 or 5
CALL P,Ø	F4ØØØØ		3 or 5
CALL PE,Ø	ECØØØØ		3 or 5
CALL PO,Ø	E4ØØØØ		3 or 5
CALL Z,Ø	CCØØØØ		3 or 5
CCF	3F	C	1
CP Ø	FEØØ	C Z P/V S	2
CP (HL)	BE	C Z P/V S	2
CP (IX+Ø)	DDBEØØ	C Z P/V S	5
CP (IY+Ø)	FDBEØØ	C Z P/V S	5
CP A	BF	C Z P/V S	1
CP B	B8	C Z P/V S	1
CP C	B9	C Z P/V S	1

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
CP D	BA	C Z P/V S	1
CP E	BB	C Z P/V S	1
CP H	BC	C Z P/V S	1
CP L	BD	C Z P/V S	1
CPD	EDA9	Z P/V S	4
CPDR	EDB9	Z P/V S	4 or 5
CPI	EDA1	Z P/V S	4
CPIR	EDB1	Z P/V S	4 or 5
CPL	2F		1
DAA	27	C Z P/V S	1
DEC (HL)	35	Z P/V S	3
DEC (IX+Ø)	DD3500	Z P/V S	6
DEC (IY+Ø)	FD3500	Z P/V S	6
DEC A	3D	Z P/V S	1
DEC B	Ø5	Z P/V S	1
DEC BC	ØB		1
DEC C	ØD	Z P/V S	1
DEC D	15	Z P/V S	1
DEC DE	1B		1
DEC E	1D	Z P/V S	1
DEC H	25	Z P/V S	1

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
DEC HL	2B		1
DEC IX	DD2B		1
DEC IY	FD2B		1
DEC L	2D	Z P/V S	1
DEC SP	3B		1
DI	F3		1
DJNZ Ø	1ØØ		2 or 3
EI	FB		1
EX (SP),HL	E3		5
EX (SP),IX	DDE3		6
EX (SP),IY	FDE3		6
EX AF,AF'	Ø8	C Z P/V S	1
EX DE,HL	EB		1
EXX	D9		1
HALT	76		1
IM Ø	ED46		2
IM 1	ED56		2
IM 2	ED5E		2
IN A,(Ø)	DBØØ		3
IN A,(C)	ED78	Z P/V S	3
IN B,(C)	ED4Ø	Z P/V S	3

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
IN C,(C)	ED48	Z P/V S	3
IN D,(C)	ED50	Z P/V S	3
IN E,(C)	ED58	Z P/V S	3
IN H,(C)	ED60	Z P/V S	3
IN L,(C)	ED68	Z P/V S	3
INC (HL)	34	Z P/V S	3
INC (IX+0)	DD3400	Z P/V S	6
INC (IY+0)	FD3400	Z P/V S	6
INC A	3C	Z P/V S	1
INC B	04	Z P/V S	1
INC BC	03		1
INC C	0C	Z P/V S	1
INC D	14	Z P/V S	1
INC DE	13		1
INC E	1C	Z P/V S	1
INC H	24	Z P/V S	1
INC HL	23	Z P/V S	1
INC IX	DD23		2
INC IY	FD23		2
INC L	2C	Z P/V S	1
INC SP	33		1

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
IND	EDAA	Z	4
INDR	EDBA		4 or 5
INI	EDA2	Z	4
INIR	EDB2		4 or 5
JP Ø	C30000		3
JP (HL)	E9		1
JP (IX)	DDE9		2
JP (IY)	FDE9		2
JP C,Ø	DA0000		3
JP M,Ø	FA0000		3
JP NC,Ø	D20000		3
JP NZ,Ø	C20000		3
JP P,Ø	F20000		3
JP PE,Ø	EA0000		3
JP PO,Ø	E20000		3
JP Z,Ø	CA0000		3
JR Ø	1800		3
JR C,Ø	3800		2 or 3
JR NC,Ø	3000		2 or 3
JR NZ,Ø	2000		2 or 3
JR Z,Ø	2800		2 or 3

MNEMONIC		OBJECT CODE	FLAGS	M-CYCLES
LD	(BC),A	02		2
LD	(DE),A	12		2
LD	(HL),0	3600		3
LD	(HL),A	77		2
LD	(HL),B	70		2
LD	(HL),C	71		2
LD	(HL),D	72		2
LD	(HL),E	73		2
LD	(HL),H	74		2
LD	(HL),L	75		2
LD	(IX+0),0	DD360000		5
LD	(IX+0),A	DD7700		5
LD	(IX+0),B	DD7000		5
LD	(IX+0),C	DD7100		5
LD	(IX+0),D	DD7200		5
LD	(IX+0),E	DD7300		5
LD	(IX+0),H	DD7400		5
LD	(IX+0),L	DD7500		5
LD	(IY+0),0	FD360000		5
LD	(IY+0),A	FD7700		5
LD	(IY+0),B	FD7000		5

MNEMONIC		OBJECT CODE	FLAGS	M-CYCLES
LD	(IY+0),C	FD7100		5
LD	(IY+0),D	FD7200		5
LD	(IY+0),E	FD7300		5
LD	(IY+0),H	FD7400		5
LD	(IY+0),L	FD7500		5
LD	(0000),A	320000		4
LD	(0000),BC	ED430000		6
LD	(0000),DE	ED530000		6
LD	(0000),HL	220000		5
LD	(0000),IX	DD220000		6
LD	(0000),IY	FD220000		6
LD	(0000),SP	ED730000		6
LD	A,(BC)	0A		2
LD	A,(DE)	1A		2
LD	A,(HL)	7E		2
LD	A,(IX+0)	DD7E00		5
LD	A,(IY+0)	FD7E00		5
LD	A,(0000)	3A0000		4
LD	A,0	3E00		2
LD	A,A	7F		1
LD	A,B	78		1

MNEMONIC		OBJECT CODE	FLAGS	M-CYCLES
LD	A,C	79		1
LD	A,D	7A		1
LD	A,H	7C		1
LD	A,I	ED57	Z P/V S	2
LD	A,L	7D		1
LD	A,R	ED5F	Z P/V S	2
LD	B,(HL)	46		2
LD	B,(IX+0)	DD4600		5
LD	B,(IY+0)	FD4600		5
LD	B,0	0600		2
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	BC,(0000)	ED4B0000		6
LD	BC,0000	010000		5
LD	C,(HL)	4E		2
LD	C,(IX+0)	DD4E00		5

MNEMONIC		OBJECT CODE	FLAGS	M-CYCLES
LD	C,(IY+Ø)	FD4EØØ		5
LD	C,Ø	ØEØØ		2
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,(HL)	56		2
LD	D,(IX+Ø)	DD56ØØ		5
LD	D,(IY+Ø)	FD56ØØ		5
LD	D,Ø	16ØØ		2
LD	D,A	57		1
LD	D,B	5Ø		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	DE,(ØØØØ)	ED5BØØØØ		6

MNEMONIC		OBJECT CODE	FLAGS	M-CYCLES
LD	DE,0000	110000		5
LD	E,(HL)	5E		2
LD	E,(IX+0)	DD5E00		5
LD	E,(IY+0)	FD5E00		5
LD	E,0	1E00		2
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,(HL)	66		2
LD	H,(IX+0)	DD6600		5
LD	H,(IY+0)	FD6600		5
LD	H,0	2600		2
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

MNEMONIC		OBJECT CODE	FLAGS	M-CYCLES
LD	H,H	64		1
LD	H,L	65		1
LD	HL,(0000)	2A0000		6
LD	HL,0000	210000		5
LD	I,A	ED47		2
LD	IX,(0000)	DD2A0000		6
LD	IX,0000	DD210000		4
LD	IY,(0000)	FD2A0000		4
LD	IY,0000	FD210000		4
LD	L,(HL)	6E		2
LD	L,(IX+0)	DD6E00		5
LD	L,(IY+0)	FD6E00		5
LD	L,0	2E00		2
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
LD	R,A	ED4F		2

MNEMONIC		OBJECT CODE	FLAGS	M-CYCLES
LD	SP,(0000)	ED7B0000		6
LD	SP,0000	310000		3
LD	SP,HL	F9		1
LD	SP,IX	DDF9		2
LD	SP,IY	FDF9		2
LDD		EDA8	P/V	4
LDDR		EDB8		4 or 5
LDI		EDA0	P/V	4
LDIR		EDB0		4 or 5
NEG		ED44	C Z P/V S	3
NOP		00		1
OR	(HL)	B6	Z P/V S	2
OR	(IX+0)	DDB600	Z P/V S	5
OR	(IY+0)	FDB600	Z P/V S	5
OR	0	F600	Z P/V S	2
OR	A	B7	Z P/V S	1
OR	B	B0	Z P/V S	1
OR	C	B1	Z P/V S	1
OR	D	B2	Z P/V S	1
OR	E	B3	Z P/V S	1
OR	H	B4	Z P/V S	1

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
OR L	B5	Z P/V S	1
OTDR	EDBB		4 or 5
OTIR	EDB3		4 or 5
OUT (C),A	ED79		3
OUT (C),B	ED41		3
OUT (C),C	ED49		3
OUT (C),D	ED51		3
OUT (C),E	ED59		3
OUT (C),H	ED61		3
OUT (C),L	ED69		3
OUT (Ø),A	D3ØØ		3
OUTD	EDAB	Z	4
OUTI	EDA3	Z	4
POP AF	F1	C Z P/V S	3
POP BC	C1		3
POP DE	D1		3
POP HL	E1		3
POP IX	DDE1		4
POP IY	FDE1		4
PUSH AF	F5		3
PUSH BC	C5		3

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
PUSH DE	D5		3
PUSH HL	E5		3
PUSH IX	DDE5		4
PUSH IY	FDE5		4
RES 0,(HL)	CB86		4
RES 0,(IX+0)	DDCB0086		6
RES 0,(IY+0)	FDCB0086		6
RES 0,A	CB87		2
RES 0,B	CB80		2
RES 0,C	CB81		2
RES 0,D	CB82		2
RES 0,E	CB83		2
RES 0,H	CB84		2
RES 0,L	CB85		2
RES 1,(HL)	CB8E		4
RES 1,(IX+0)	DDCB008E		6
RES 1,(IY+0)	FDCB008E		6
RES 1,A	CB8F		2
RES 1,B	CB88		2
RES 1,C	CB89		2
RES 1,D	CB8A		2

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
RES 1,E	CB8B		2
RES 1,H	CB8C		2
RES 1,L	CB8D		2
RES 2,(HL)	CB96		4
RES 2,(IX+Ø)	DDCBØØ96		6
RES 2,(IY+Ø)	FDCBØØ96		6
RES 2,A	CB97		2
RES 2,B	CB9Ø		2
RES 2,C	CB91		2
RES 2,D	CB92		2
RES 2,E	CB93		2
RES 2,H	CB94		2
RES 2,L	CB95		2
RES 3,(HL)	CB9E		4
RES 3,(IX+Ø)	DDCBØØ9E		6
RES 3,(IY+Ø)	FDCBØØ9E		6
RES 3,A	CB9F		2
RES 3,B	CB98		2
RES 3,C	CB99		2
RES 3,D	CB9A		2
RES 3,E	CB9B		2

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
RES 3,H	CB9C		2
RES 3,L	CB9D		2
RES 4,(HL)	CBA6		4
RES 4,(IX+Ø)	DDCBØØA6		6
RES 4,(IY+Ø)	FDCBØØA6		6
RES 4,A	CBA7		2
RES 4,B	CBA0		2
RES 4,C	CBA1		2
RES 4,D	CBA2		2
RES 4,E	CBA3		2
RES 4,H	CBA4		2
RES 4,L	CBA5		2
RES 5,(HL)	CBAE		4
RES 5,(IX+Ø)	DDCBØØAE		6
RES 5,(IY+Ø)	FDCBØØAE		6
RES 5,A	CBAF		2
RES 5,B	CBA8		2
RES 5,C	CBA9		2
RES 7,L	CBBD		2
RET	C9		3
RET C	D8		1 or 3

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
RET M	F8		1 or 3
RET NC	D0		1 or 3
RET NZ	C0		1 or 3
RET P	F0		1 or 3
RET PE	E8		1 or 3
RET PO	E0		1 or 3
RET Z	C8		1 or 3
RETI	ED4D		4
RETN	ED45		4
RL (HL)	CB16	C Z P/V S	4
RL (IX+0)	DDCB0016	C Z P/V S	6
RL (IY+0)	FDCB0016	C Z P/V S	6
RL A	CB17	C Z P/V S	2
RL B	CB10	C Z P/V S	2
RL C	CB11	C Z P/V S	2
RL D	CB12	C Z P/V S	2
RL E	CB13	C Z P/V S	2
RL H	CB14	C Z P/V S	2
RL L	CB15	C Z P/V S	2
RLA	17	C	1
RES 5,D	CBAA		2

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
RES 5,E	CBAB		2
RES 5,H	CBAC		2
RES 5,L	CBAD		2
RES 6,(HL)	CBB6		4
RES 6,(IX+Ø)	DDCBØØB6		6
RES 6,(IY+Ø)	FDCBØØB6		6
RES 6,A	CBB7		2
RES 6,B	CBBØ		2
RES 6,C	CBB1		2
RES 6,D	CBB2		2
RES 6,E	CBB3		2
RES 6,H	CBB4		2
RES 6,L	CBB5		2
RES 7,(HL)	CBBE		4
RES 7,(IX+Ø)	DDCBØØBE		6
RES 7,(IY+Ø)	FDCBØØBE		6
RES 7,A	CBBF		2
RES 7,B	CBB8		2
RES 7,C	CBB9		2
RES 7,D	CBBA		2
RES 7,E	CBBB		2

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
RES 7,H	CBBC		2
RLC (HL)	CB06	C Z P/V S	4
RLC (IX+0)	DDCB0016	C Z P/V S	6
RLC (IY+0)	FDCB0016	C Z P/V S	6
RLC A	CB07	C Z P/V S	2
RLC B	CB00	C Z P/V S	2
RLC C	CB01	C Z P/V S	2
RLC D	CB02	C Z P/V S	2
RLC E	CB03	C Z P/V S	2
RLC H	CB04	C Z P/V S	2
RLC L	CB05	C Z P/V S	2
RLCA	07	C	1
RLD	ED6F	Z P/V S	5
RR (HL)	CB1E	C Z P/V S	4
RR (IX+0)	DDCB001E	C Z P/V S	6
RR (IY+0)	FDCB001E	C Z P/V S	6
RR A	CB1F	C Z P/V S	2
RR B	CB18	C Z P/V S	2
RR C	CB19	C Z P/V S	2
RR D	CB20	C Z P/V S	2
RR E	CB21	C Z P/V S	2

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
RR H	CB22	C Z P/V S	2
RR L	CB23	C Z P/V S	2
RRA	1F	C	1
RRC (HL)	CB0E	C Z P/V S	4
RRC (IX+0)	DDCB000E	C Z P/V S	6
RRC (IY+0)	FDCB000E	C Z P/V S	6
RRC A	CB0F	C Z P/V S	2
RRC B	CB08	C Z P/V S	2
RRC C	CB09	C Z P/V S	2
RRC D	CB0A	C Z P/V S	2
RRC E	CB0B	C Z P/V S	2
RRC H	CB0C	C Z P/V S	2
RRC L	CB0D	C Z P/V S	2
RRCA	0F	C	1
RRD		Z P/V S	5
RST 00H	C7		3
RST 08H	CF		3
RST 10H	D7		3
RST 18H	DF		3
RST 20H	E7		3
RST 28H	EF		3

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
RST 30H	F7		3
RST 38H	FF		3
SBC A,0	DE00	C Z P/V S	2
SBC A,(HL)	9E	C Z P/V S	2
SBC A,(IX+0)	DD9E00	C Z P/V S	5
SBC A,(IY+0)	FD9E00	C Z P/V S	5
SBC A,A	9F	C Z P/V S	1
SBC A,B	98	C Z P/V S	1
SBC A,C	99	C Z P/V S	1
SBC A,D	9A	C Z P/V S	1
SBC A,E	9B	C Z P/V S	1
SBC A,H	9C	C Z P/V S	1
SBC A,L	9D	C Z P/V S	1
SBC HL,BC	ED42	C Z P/V S	2
SBC HL,DE	ED52	C Z P/V S	2
SBC HL,HL	ED62	C Z P/V S	2
SBC HL,SP	ED72	C Z P/V S	2
SCF	37	C	1
SET 0,(HL)	CBC6		4
SET 0,(IX+0)	DDCB00C6		6
SET 0,(IY+0)	FDCB00C6		6

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
SET 0,A	CBC7		2
SET 0,B	CBC0		2
SET 0,C	CBC1		2
SET 0,D	CBC2		2
SET 0,E	CBC3		2
SET 0,H	CBC4		2
SET 0,L	CBC5		2
SET 1,(HL)	CBCE		4
SET 1,(IX+0)	DDCB00CE		6
SET 1,(IY+0)	FDCB00CE		6
SET 1,A	CBCF		2
SET 1,B	CBC8		2
SET 1,C	CBC9		2
SET 1,D	CBCA		2
SET 1,E	CBCB		2
SET 1,H	CBCC		2
SET 1,L	CBCD		2
SET 2,(HL)	CBD6		4
SET 2,(IX+0)	DDCB00D6		6
SET 2,(IY+0)	FDCB00D6		6
SET 2,A	CBD7		2

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
SET 2,B	CBD0		2
SET 2,C	CBD1		2
SET 2,D	CBD2		2
SET 2,E	CBD3		2
SET 2,H	CBD4		2
SET 2,L	CBD5		2
SET 3,(HL)	CBDE		4
SET 3,(IX+0)	DDCB00DE		6
SET 3,(IY+0)	FDCB00DE		6
SET 3,A	CBDF		2
SET 3,B	CBD8		2
SET 3,C	CBD9		2
SET 3,D	CBDA		2
SET 3,E	CBDB		2
SET 3,H	CBDC		2
SET 3,L	CBDD		2
SET 4,(HL)	CBE6		4
SET 4,(IX+0)	DDCB00E6		6
SET 4,(IY+0)	FDCB00E6		6
SET 4,A	CBE7		2
SET 4,B	CBE0		2

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
SET 4,C	CBE1		2
SET 4,D	CBE2		2
SET 4,E	CBE3		2
SET 4,H	CBE4		2
SET 4,L	CBE5		2
SET 5,(HL)	CBEE		4
SET 5,(IX+Ø)	DDCBØØEE		6
SET 5,(IY+Ø)	FDCBØØEE		6
SET 5,A	CBEF		2
SET 5,B	CBE8		2
SET 5,C	CBE9		2
SET 5,D	CBEA		2
SET 5,H	CBEB		2
SET 5,L	CBEC		2
SET 6,(HL)	CBF6		4
SET 6,(IX+Ø)	DDCBØØF6		6
SET 6,(IY+Ø)	FDCBØØF6		6
SET 6,A	CBF7		2
SET 6,B	CBFØ		2
SET 6,C	CBF1		2
SET 6,D	CBF2		2

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
SET 6,E	CBF3		2
SET 6,H	CBF4		2
SET 6,L	CBF5		2
SET 7,(HL)	CBFE		4
SET 7,(IX+0)	DDCB00FE		6
SET 7,(IY+0)	FDCB00FE		6
SET 7,A	CBFF		2
SET 7,B	CBF8		2
SET 7,C	CBF9		2
SET 7,D	CBFA		2
SET 7,E	CBFB		2
SET 7,H	CBFC		2
SET 7,L	CBFD		2
SLA (HL)	CB26	C Z P/V S	4
SLA (IX+0)	DDCB0026	C Z P/V S	6
SLA (IY+0)	FDCB0026	C Z P/V S	6
SLA A	CB27	C Z P/V S	2
SLA B	CB20	C Z P/V S	2
SLA C	CB21	C Z P/V S	2
SLA D	CB22	C Z P/V S	2
SLA E	CB23	C Z P/V S	2

MNEMONIC	OBJECT CODE	FLAGS	M-CYCLES
SLA H	CB24	C Z P/V S	2
SLA L	CB25	C Z P/V S	2
SRA (HL)	CB2E	C Z P/V S	4
SRA (IX+Ø)	DDCBØØ2E	C Z P/V S	6
SRA (IY+Ø)	FDCBØØ2E	C Z P/V S	6
SRA A	CB2F	C Z P/V S	2
SRA B	CB28	C Z P/V S	2
SRA C	CB29	C Z P/V S	2
SRA D	CB2A	C Z P/V S	2
SRA E	CB2B	C Z P/V S	2
SRA H	CB2C	C Z P/V S	2
SRA L	CB2D	C Z P/V S	2
SRL (HL)	CB3E	C Z P/V S	4
SRL (IX+Ø)	DDCBØØ3E	C Z P/V S	6
SRL (IY+Ø)	FDCBØØ3E	C Z P/V S	6
SRL A	CB3F	C Z P/V S	2
SRL B	CB38	C Z P/V S	2
SRL C	CB39	C Z P/V S	2
SRL D	CB3A	C Z P/V S	2
SRL E	CB3B	C Z P/V S	2
SRL H	CB3C	C Z P/V S	2
SRL L	CB3D	C Z P/V S	2

SUB	Ø	D6ØØ	C Z P/V S	2
SUB	(HL)	96	C Z P/V S	2
SUB	(IX+Ø)	DD96ØØ	C Z P/V S	5
SUB	(IY+Ø)	FD96ØØ	C Z P/V S	5
SUB	A	97	C Z P/V S	1
SUB	B	9Ø	C Z P/V S	1
SUB	C	91	C Z P/V S	1
SUB	D	92	C Z P/V S	1
SUB	E	93	C Z P/V S	1
SUB	H	94	C Z P/V S	1
SUB	L	95	C Z P/V S	1
XOR	(HL)	AE	Z P/V S	2
XOR	(IX+Ø)	DDAEØØ	Z P/V S	5
XOR	(IY+Ø)	FDAEØØ	Z P/V S	5
XOR	Ø	EEØØ	Z P/V S	2
XOR	A	AF	Z P/V S	1
XOR	B	A8	Z P/V S	1
XOR	C	A9	Z P/V S	1
XOR	D	AA	Z P/V S	1
XOR	E	AB	Z P/V S	1
XOR	H	AC	Z P/V S	1
XOR	L	AD	Z P/V S	1

APPENDIX G

SUMMARY OF FLAG OPERATIONS

Symbols used:

?	all addressing modes valid for that instruction
??	all addressing modes for 16-bit data
b	bit 0-7
R	8-bit register valid for that instruction
rr	register-pair valid for that instruction
!	affected by operation
.	unaffected by operation
P	parity
O	overflow
r	reset
s	set
c	see comment

(otherwise effect unknown or unimportant).

INSTRUCTION	FLAG: Z	C	S	P/O	H	N	Comments
ADD A,? ADC A,?	!	!	!	O	!	r	
SUB ? SBC A, ? CP ? NEG	!	!	!	O	!	s	
AND ?	!	!	!	P	s	r	
OR ? XOR ?	!	r	!	P	r	r	
INC ?	!	.	!	O	!	r	
DEC ?	!	.	!	O	!	s	
ADD rr,??	.	!	!	.		r	
ADC HL,??	!	!	!	O		r	
SBC HL,??	!	!	!	O		s	
RLA RLCA RRA RRCA	.	!	.	.	r	r	
RL ? RLC ? RR ? RRC ? SLA ? SRA ? SRL ?	!	!	!	P	r	r	
RLD RRD	!	.	!	P	r	r	
DAA	.	.	.	P	!		
CPL	.	.	.	.	s	s	
SCF	.	s	.	.	r	r	
CCF	.	!	.	.		r	
IN R,(C)	!	.	!	P	r	r	
INI IND OUTI OUTD	!	.				s	
INIR INDR OTIR OTDR	s	.				s	Z = 0 if B ≠ 0
LDI LDD		.		!	r	r	
LDIR LDDR		.			r	r	P/V = 1 if BC ≠ 0
CPI CPIR CPDR	!	.		!		s	IF A = 1 Z set. P/V set if BC ≠ 0
LD A,I LD A,R	!	.	!	c	r	r	interrupt enable flip-flop copied into P/V
BIT B,?	!	.			s	r	transfers bit b of ? to Z

INDEX

* *denotes reference to a diagram*
ff. further reference found on following pages

This book fills a serious gap in literature on programming the Z80. Rather than dealing with the elementary concepts of Z80 architecture, Nitschke's book examines advanced, serious and practical machine language programming.

It has been designed essentially for three types of programmers:
— those who know the rudiments of Z80 programming and now want to tackle the more complex aspects of the instruction set in greater detail
— intermediate programmers who want to develop professional software
— experts who want to view new methods, programming styles and algorithms, to add to their library of programming knowledge.

The core of the book looks at popular programming applications. Starting with the basics of organising information, generating arrays and tables, block moving, shifting and erasing, string manipulation, data compression and command tables, it moves on to advanced applications like word and sentence decoding. This section of the book ends with a detailed analysis of an 'adventure game'.

Game programming is then examined in detail. Topics covered include random number generating, the creation of mazes in two and three dimensions, animation, arcade game development and other essential techniques.

From there, the book turns to more 'serious' applications, including binary sorts and block searching, and the concepts of database management. The chapter ends with an examination of a pseudo-computer language, to help you develop your own high-level languages. Nitschke also discusses machine language editing, break-pointing, single-stepping and disassembling. Another chapter explains how to synthesise speech and sound effects, and includes a complete set of 'undocumented' Z80 instructions.

The book also explains how to write a program for commercial publication, what is expected from such software, how to document it for submission, and what returns you can expect.

ISBN 0-907563-90-2

£12.95

ADVANCED Z80 MACHINE CODE PROGRAMMING

William Nitschke

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/