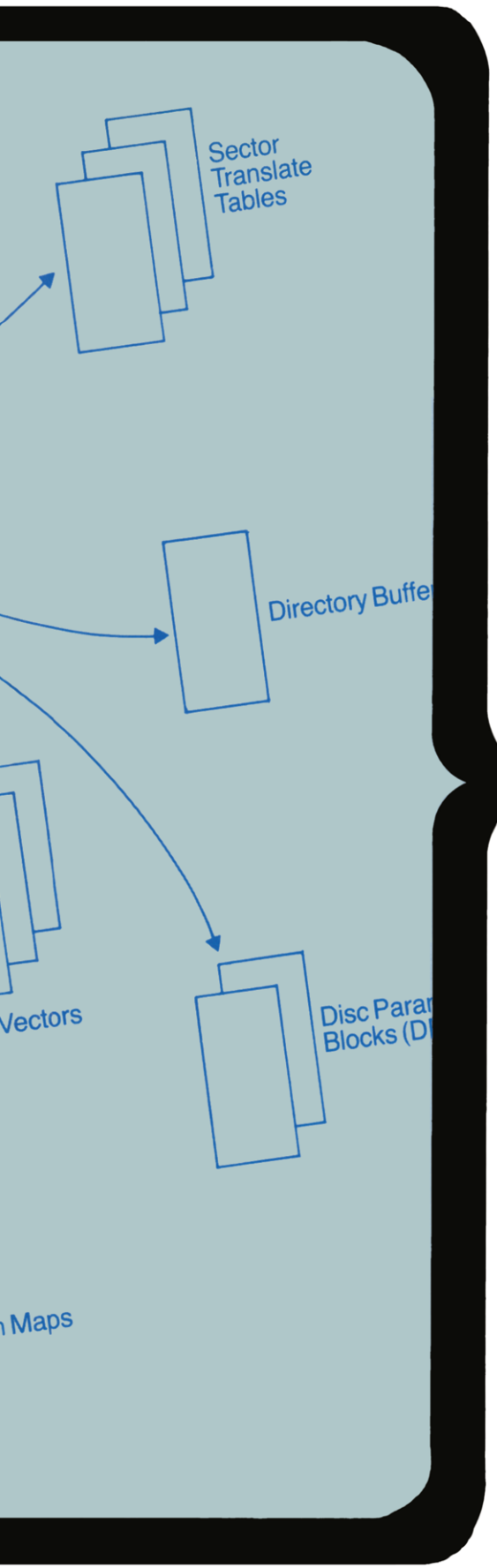


CP/M 80 Programmer's Guide

Barry Morrell
and
Peter Whittle

CP/M 80 Programmer's Guide

Macmillan Computer Science Series

Consulting Editor
Professor F. H. Sumner, University of Manchester

S. T. Allworth, *Introduction to Real-time Software Design*
Ian O. Angell, *A Practical Introduction to Computer Graphics*
R. E. Berry and B. A. E. Meekings, *A Book on C*
G. M. Birtwistle, *Discrete Event Modelling on Simula*
T. B. Boffey, *Graph Theory in Operations Research*
Richard Bornat, *Understanding and Writing Compilers*
J. K. Buckle, *Software Configuration Management*
W. D. Burnham and A. R. Hall, *Prolog Programming and Applications*
J. C. Cluley, *Interfacing to Microprocessors*
Robert Cole, *Computer Communications*
Derek Coleman, *A Structured Programming Approach to Data*
Andrew J. T. Colin, *Fundamentals of Computer Science*
Andrew J. T. Colin, *Programming and Problem-solving in Algol 68*
S. M. Deen, *Fundamentals of Data Base Systems*
S. M. Deen, *Principles and Practice of Database Systems*
P. M. Dew and K. R. James, *Introduction to Numerical Computation in Pascal*
M. R. M. Dunsmuir and G. J. Davies, *Programming the UNIX System*
K. C. E. Gee, *Introduction to Local Area Computer Networks*
J. B. Gosling, *Design of Arithmetic Units for Digital Computers*
Roger Hutty, *Fortran for Students*
Roger Hutty, *Z80 Assembly Language Programming for Students*
Roland N. Ibbett, *The Architecture of High Performance Computers*
Patrick Jaulent, *The 68000 – Hardware and Software*
J. M. King and J. P. Pardoe, *Program Design Using JSP – A Practical Introduction*
H. Kopetz, *Software Reliability*
E. V. Krishnamurthy, *Introduction Theory of Computer Science*
V. P. Lane, *Security of Computer Based Information Systems*
Graham Lee, *From Hardware to Software – an introduction to computers*
A. M. Lister, *Fundamentals of Operating Systems, third edition*
G. P. McKeown and V. J. Rayward-Smith, *Mathematics for Computing*
Brian Meek, *Fortran, PL/1 and the Algols*
Derrick Morris, *System Programming Based on the PDP11*
P. Oets, *Using MS-DOS and PC-DOS*
Christian Queinnec, *LISP*
John Race, *Case Studies in Systems Analysis*
W. P. Salman, O. Tisserand and B. Toulout, *FORTH*
L. E. Scales, *Introduction to Non-linear Optimization*
Peter S. Sell, *Expert Systems – A Practical Introduction*
Colin J. Theaker and Graham R. Brookes, *A Practical Course on Operating Systems*
M. J. Usher, *Information Theory for Information Technologists*
B. S. Walker, *Understanding Microprocessors*
Peter J. L. Wallis, *Portable Programming*
I. R. Wilson and A. M. Addyman, *A Practical Introduction to Pascal – with BS6192, second edition*

Other Macmillan titles of related interest

J. E. Bingham and G. W. P. Davies, *A Handbook of Systems Analysis, second edition*
J. E. Bingham and G. W. P. Davies, *Planning for Data Communications*

CP/M 80 Programmer's Guide

Barry Morrell
B.Sc.

Peter Whittle
B.Sc.

MACMILLAN

© Barry Morrell and Peter Whittle 1985

All rights reserved. No reproduction, copy or transmission of this publication may be made without written permission.

No paragraph of this publication may be reproduced, copied or transmitted save with written permission or in accordance with the provisions of the Copyright Act 1956 (as amended).

Any person who does any unauthorised act in relation to this publication may be liable to criminal prosecution and civil claims for damages.

First published 1985

Published by
MACMILLAN EDUCATION LTD
Houndmills, Basingstoke, Hampshire RG21 2XS
and London
Companies and representatives
throughout the world

Typeset by TecSet Ltd, Sutton, Surrey

British Library Cataloguing in Publication Data
Morrell, Barry
CP/M 80 programmer's guide.
1. CP/M-80 (Computer operating system)
I. Title II. Whittle, Peter
001.64'25 QA76.6

ISBN 978-0-333-39558-5 ISBN 978-1-349-08123-3 (eBook)
DOI 10.1007/978-1-349-08123-3

Contents

Preface *ix*
Acknowledgements *x*

1 Introduction **1**
What is an operating system? 1
What is CP/M 80? 2
What does CP/M 80 consist of? 3
How do you interface with CP/M 80? 4
How do your programs interface with CP/M 80? 4
Why use machine code? 6
Using high level languages under CP/M 6
Summary 6

2 Program Development **8**
Design 8
Program documentation 9
Cutting code 9
Assembling, loading and running 10
Basic principles 10
Building libraries 13
Testing and debugging 14
Optimisation 14
Summary 15

3 Design **16**
Information gathering 16
The main design activity 17
General points 17
Subroutine coupling 18
Program portability 19
The final stages of design 20
Summary 21

4 Using Simple BDOS Functions **22**
The BDOS function call 22

Simple output using BDOS functions 24
Simple input using BDOS functions 26
A more efficient mode of input 28
Bypassing the BDOS 30
Summary 34

5 File Handling 36
Discs and files 36
Creating a simple file 38
Reading a simple file 41
Adding data to the end of a file 43
More about files 45
FCBs and their contents 45
Sectors and blocking 45
Types of file 47
Data areas 47
Passing filenames via the keyboard 49
Directory operations 51
Using random access with files 55
Random access of a sequentially-written file 56
Writing sectors in random order 58
Error handling with files 60
Summary 60

6 Disc Operations 62
Introduction 62
Protecting your discs and files 63
Getting hold of disc characteristics 64
Introduction to the disc data structures 64
The allocation maps 67
The Disc Parameter Blocks (DPBs) 68
Allocation block sizes 70
Summary 71

7 Debugging your Programs 73
General principles 73
How do bugs get into programs? 73
Keeping bugs out of your programs 74
Trapping the ones that got away! 74
Getting rid of the rest by testing 75
Debugging under CP/M 76
Loading a program (or ZSID) from disc 77
Setting breakpoints and running the code 77
Displaying the register set 78

Displaying an area of memory 78
'Patching' data 79
'Patching' a program 80
Disassembling a piece of code 80
Summary 81

8 Random Access Files **82**
Creating sequential files 82
How random access files are allocated 84
Summary 86

9 Using BDOS Functions from High Level Languages **87**
Accessing machine code from BASIC 88
Accessing machine code from Pascal 89
Summary 91

Appendix A: Example Programs **92**
Reaction timer program 92
File size calculation program 92
Disc free space program 92

Appendix B: CP/M Memory Map *106*

Appendix C: ASCII Code Tables *107*

Appendix D: BDOS Functions *109*
BDOS calling mechanism *109*
Summary of BDOS functions *109*

Appendix E: BDOS Functions of CP/M 80 Family *155*

Appendix F: The ZASM Macro Assembler *158*

Index *159*

Preface

This book is for people who want to learn how to program under the CP/M 80 Operating System. It is *not* a guide to using CP/M. Before reading it you should have some knowledge of programming using Z80 machine code and programming in a high level language such as BASIC. You should also have read a book on CP/M operating such as *Using CP/M* by Peter Gosling (Macmillan, 1985).

The main part of the present book is meant to be read sequentially and, if you have little experience of machine code programming, this is the way that you should read it. If you are already an experienced programmer, you may want to 'dive' into the book at a particular point. In this case, you will find a list of keywords at the front of each chapter which will tell you what is covered within that chapter, and a more detailed summary at the end of most chapters.

The appendixes are meant to be used for reference purposes. They include descriptions of BDOS functions, laid out in alphabetical order with a separate page for each function. A list of functions grouped under appropriate headings is also given.

Examples in this book are written using Research Machines Ltd's Z80 macro assembler ZASM, mainly because it is a true Z80 assembler. However, with minor changes they can run under other assemblers such as Microsoft's M80. The main differences between ZASM and M80 are summarised in appendix F.

Acknowledgements

To reviewers and Research Machines Ltd for their help. In particular, to Research Machines for their permission to use some of the information on program portability in chapter 3.

CP/M and CP/NET are registered trademarks of Digital Research. Zilog and Z80 are trademarks of Zilog Inc.

ProPascal is a registered trademark of Prospero Software.

Microsoft BASIC is a registered trademark of Microsoft.

ZASM is a registered trademark of Research Machines Ltd.

Wordstar is a registered trademark of MicroPro International Corporation.

1 Introduction

Application programs	*Interfaces*
BDOS	*Mixed language programming support*
BDOS functions	*Machine code*
BIOS	*Operating system*
CCP	*Optimisation*
Firmware	*TPA*
High level languages	

CP/M is an *operating system* which was first produced in 1975 by Gary Kildall, a consultant at Intel Corporation, and its name stands for *C*ontrol *P*rogram for *M*icrocomputers. It was first used on the 8-bit 8080 microprocessor and subsequently ran on a number of different computers.

What is an operating system?

Until recently, computer programming was regarded as a black art by most people. With the expanding use of home computers this image has changed. However, even when you know how to program a computer there is still an area that has the image of an 'inner sanctum'. This is the operating system.

An operating system like CP/M is nothing more than a sequence of instructions to your computer – similar to your programs, in fact. It transforms the hardware of your computer into something convenient and easy to use and provides a framework within which you can run your application programs.

In order to understand the need for an operating system, imagine that you want to build a house. If you live in a part of the world where there are no bricks, you would have to make your own 'building blocks'. This being the case, you might as well make some that are the optimum size and shape to your requirements. On the other hand, if bricks are easy to get hold of you should use them: they may not be exactly what you want, but by using them you will save yourself a lot of work. If you start off with a new, 'empty' computer, you have a similar problem. Your first program is likely to contain instructions which either input data, output it, or do both. If you know that you will only want to write this *one* program, you can write special input/output instructions and optimise them for your program. However, if you think you will write some more programs, it

would be sensible to put these instructions into subroutines and make them general enough to be of use in other programs.

Now, suppose you want to sell the computer to a lot of other people who will write their own programs. It would be sensible to make your subroutines available to them as well, in the form of a package. In fact,this is what happens and the package is called an operating system. There are other parts to an operating system besides the input/output subroutines. The main ones are as follows

- Some *utilities,* which are basically just programs that perform standard tasks such as copying data from one device to another
- A means of communicating with the operating system from the keyboard using commands, otherwise known as a *command processor*

Operating systems should provide at least one other facility: the basis for error handling and recovery when things go wrong. Earlier versions of CP/M were lacking in this respect; however, Personal CP/M and CP/M Plus give you much more control when errors such as disc failures occur.

More sophisticated operating systems (including some versions of CP/M) provide you with other facilities. For example, some allow you to run a number of programs at the same time ('concurrently'). However, they all have one thing in common: they provide the programmer with a consistent set of building blocks to help him develop his programs.

What is CP/M 80?

There are now several million copies of CP/M in the field and it exists in a number of versions aimed at different machines and markets. These fall into three main families

- CP/M 80, which runs on Intel 8080-compatible computers, including those using the more popular Zilog Z80 chip
- CP/M 68000, which runs on computers using the Motorola 68000 processor
- CP/M 86, which runs on computers using the 8086 family of processors

This book describes how to write programs using the main versions of CP/M 80: CP/M 1.4 and CP/M 2.2. It does *not* describe the additional facilities of CP/M Plus 80 or Personal CP/M 80. However, the general principles apply to these as well and this book should be a useful introduction to CP/M before you look at either of these operating systems. Examples are given in Zilog Z80 machine code because most versions of CP/M 80 are used with Z80 processors. You should therefore have an understanding of Z80 machine code.

Appendix E summarises the differences between the various versions of CP/M. For the present, all you need to understand is that CP/M is a control program, or operating system.

What does CP/M 80 consist of?

CP/M is divided into four parts

- The *BIOS* (*B*asic *I*nput/*O*utput *S*ystem)
- The *BDOS* (*B*asic *D*isc *O*perating *S*ystem)
- The *CCP* (*C*onsole *C*ommand *P*rocessor)
- The *TPA* (*T*ransient *P*rogram *A*rea), the area into which your own programs fit

Their position in your computer's memory is shown in figure 1.1. This also shows the names for some of the more useful addresses.

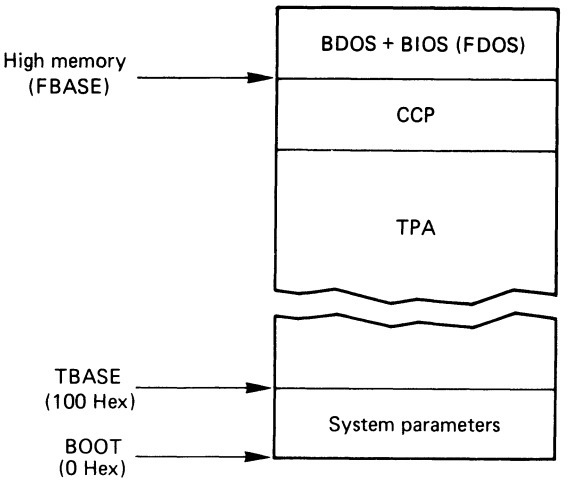


Figure 1.1 Organisation of CP/M

The BIOS defines the low level interface with your computer system which is necessary for device input/output. It is different for every make of computer. The BDOS defines file structure for discs and handles access to devices such as the keyboard and screen. Both are combined into a single module with a common entry point from your programs; this is referred to as the *FDOS.*

The CCP acts as an interface between the computer and yourself. It analyses whatever you type on the keyboard (for example, TYPE, DIR) and asks the FDOS to carry out the operations requested.

The TPA is a fancy name for an area where you can load and run programs. It is also used by CP/M to hold system utilities such as PIP during their period of operation. You can increase the size of the TPA to include the area covered by the CCP, as long as you reload the CCP when your program has finished.

Digital Research refer to user programs as transient programs. We will not use this unfriendly name; instead, we will call them *your programs.*

How do you interface with CP/M 80?

You communicate with CP/M via the CCP by typing *command lines* such as DIR and REN after each prompt. Each command line takes one of the following three forms

command
command file1
command file1 file2

command can be one of three things: a built-in function held within CP/M such as DIR and TYPE; the name of a command such as PIP, which is loaded into the TPA; or a program that you have written yourself. In each case, CP/M will try to run the command or program, loading it if necessary from disc.

Suppose the following form of command line is used

command

If the command is a built-in function of CP/M, it is executed immediately. Otherwise, the CCP searches the currently addressed disc for a file with the name *command.COM* which contains a program. If it finds one, the program will be loaded.

The other two types of command line (the ones containing *file1, file2*) allow you to communicate file information to your program from the keyboard. In these cases, the CCP loads the command or program into memory as before and makes the file description(s) available to it. The method used by programs to extract the information is described in Chapter 5.

The CCP now passes control to the command or program and this begins execution.

How do your programs interface with CP/M 80?

CP/M has a number of interfaces and these are shown in figure 1.2. The layers shown normally communicate with the next layer down; for example, your programs normally communicate with the BDOS. However, there are exceptions, and under special circumstances your programs may wish to communicate with lower layers.

At the highest level is the interface between yourself and your *application*

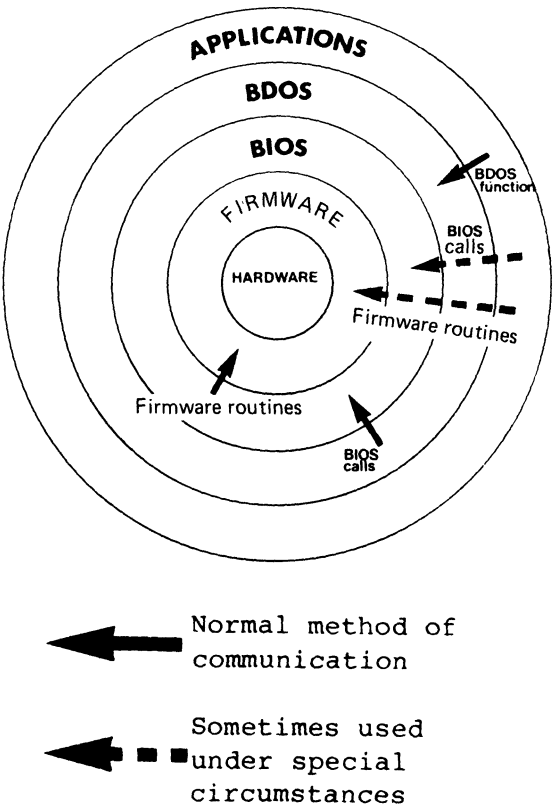


Figure 1.2 CP/M interfaces

programs. Below this is the interface between your programs and the BDOS. Your programs can access this interface using *BDOS functions* and this is the normal method of programming under CP/M. Chapters 4, 5 and 6 tell you how to write programs using BDOS functions and the functions themselves are described in detail in appendix D.

At the next level down is the interface between your programs and the BIOS. Your programs can access this interface, but programming at this level requires a special knowledge of the BIOS itself and is not described in this book.

At the lowest level, your programs may be able to access the facilities of the computer's own 'operating system', or *firmware.* This may or may not be described in the manuals of your computer. If it is, you should access it only if you have no alternative; firmware requires a lot of understanding and careful use, and programs that access it directly are not easily transportable from one type of computer to another.

Why use machine code?

Machine code programming is another of computing's sacred cows. Most of the time you should not need to do it and, in fact, it is better if you do not. Programming in machine code is a slow process. It tends to be inefficient in human resources when you can do the job adequately in a high level language such as PASCAL or BASIC.

Machine code is in the form that computers understand best, but it is not easily understood by human beings. There are two main reasons for using it: to produce either faster or more compact code. Arcade games tend to be written in machine code because they need to be fast. Operating systems like CP/M are often written in it because they need to be as compact and fast as possible.

Programs can be written in machine code from the outset, or they can first be written in a high level language and then converted to machine code. In the second case, when you convert a high level language program to machine code you do not always need to rewrite the entire program. A more efficient solution is to concentrate on areas that will give the greatest improvement in the shortest possible programming time. This area of programming is known as *optimisation* and it is covered in more detail in chapter 2.

Using high level languages under CP/M

High level languages such as PASCAL and BASIC sometimes have standard subroutines which allow you to access the operating system. Using them you could, for example, determine the operating system version number or read the disc directory.

Even if there are no standard subroutines available, the high level language might let you write your own routines in another language (including machine code) and 'bolt' them on to your high level language program. This is called *mixed language programming support* and it is described in more detail in chapter 9.

Summary

- By now, you should understand what an operating system like CP/M is, even if you do not know in detail how it works. You should also know the main parts of CP/M.
- Perhaps the most important thing to understand before you go any further are the reasons for using machine code and high level languages and, indeed, *how* you use them. If you need further explanations about machine code, try looking through either of the following books before you go any further:

- *Z80 Assembly Language Programming for Students* by Roger Hutty (Macmillan, 1981)
- *Programming in Z80 Assembly Language* by Roger Hutty (Macmillan, 1984)

If you already have this knowledge, read on!

2 *Program Development*

2 Program Development		
Assembler	*Documentation*	
Assembling	*Libraries*	*Running*
Coding	*Loading*	*Testing*
Debugging	*Optimisation*	*Text editor*
Design	*Program source file*	*Zasm*

This chapter covers the main stages of program development. Some of the areas covered may seem incongruous to the home programmer. However, the boundary between the professional and the amateur programmer is blurred these days, when home enthusiasts are selling their wares.

Even if financial reward is not the end point, it is still worth adopting a professional approach to program development. You may save time in the short term by adopting a sloppy approach, but many hours may be lost in the long term.

There are six main stages of program development

- Design
- Program documentation
- Coding
- Assembling, loading and running
- Testing and debugging
- Optimisation

The rest of this chapter covers each of these areas in turn.

Design

Programs that are 'thrown' together are likely to be less reliable than those that are designed. The bugs that they contain can also take a long time to 'iron out'. If the program is small this may not matter. If it is sophisticated a lot of time will be lost.

Much more time should be spent on the design than on the actual coding; probably two-thirds on design to one-third on coding. What often happens is the

reverse: something like 2 days for the design, 2 weeks of coding, 1 month of debugging and 2 months of redesign!

Design is one of the most important areas of program development. For this reason, an entire chapter has been devoted to it – see chapter 3.

Program documentation

The importance of good *program* documentation (as opposed to *user* documentation) is realised only when you are modifying a program some time after it has been written. Even if you wrote the original program yourself, a lack of documentation can still cause problems as you will have probably forgotten how it works. If someone else wrote it the problem is much bigger.

The best way to regard program documentation is this: if you make it good enough for other people to follow easily, they will not need to bother you when they are modifying your program. You can then get on with something more interesting.

You should write program documentation as you are developing the program. This has two immediate benefits

- It consolidates your idea of how the program works
- Bugs are easier to track down during program development

If you have not been documenting as you go along, whenever you get a section of code working, go back and document it. Do not postpone documentation until you have done more coding; by then, you will have forgotten the points that are worth documenting.

If you keep detailed design notes of any tabular structures you can modify them when necessary and use them as a sizeable part of the final documentation. It is best to keep them in a proper notebook: loose pages can easily be mislaid or get out of sequence. However, if time is short, throw a few blank pages at the end of each assembler listing and scribble notes on them.

There is one other point to mention in connection with screen documentation. If you create screen pictures at the design stage using a text editor, or by some other mechanism, you can get initial feedback from potential users and get someone else to document them while you are producing the software.

Cutting code

Programming is an exact science and there is a strong need for clear thinking, especially when coding. A piece of code that is 'fuzzy' and difficult to follow is more likely to contain bugs. Code that is clear, concise and easy to read is more likely to work correctly.

When you first code a program you should concentrate on clarity. Don't worry about the program's performance at this stage: you can optimise the implementation later, when the program is up and running. The most effective contribution towards performance should have been made at the design stage, by selecting the best algorithms. When you first code the design you should concentrate on implementing it in the *clearest* possible way.

If you try to use 'clever' code from the start it will probably take you longer to get the performance required, and the program will be more difficult for other people to maintain. Always start off by assuming that someone else will maintain your programs and that you will maintain other people's; that is a good guide-line – if you are sloppy, this will give them no incentive to be clear and you will lose in the long run.

While you are coding you should always try to use existing libraries of routines where possible: there is not much point in re-inventing the wheel. If libraries are non-existent, try building up your own for future use. For example, if you write code to parse a command line (interpret the items in it), make the code more general so that it can be used in other programs. This gives a higher cost initially, but in the long run you save a lot of time and all your programs will be similar.

One final point to remember when coding is to make your code as portable as possible. If you can isolate the areas of code that are machine-dependent, you will make the program easier to transfer on to another system. Program portability is covered further in chapter 3.

Assembling, loading and running

Basic principles

Machine code programs are in a form that the hardware can easily obey: a set of binary numbers. However, they are not very easy to input into the computer and they are certainly not easy for a human to understand.

To make programming in machine code a lot easier, you can use a program called an *assembler.* This converts text that is easier for *you* to understand into binary code that the *computer* can obey (a process called *assembly*). It also generates a listing file which you can use when you debug the program.

Before you use an assembler, you must create a text file (the *program source file*) which the assembler can understand. You do this using a *text editor.*

The rest of this section describes how to assemble, load and run a program by using a simple program example. The stages involved are summarised in figure 2.1. and parts of this figure are used in the rest of this chapter to highlight each stage.

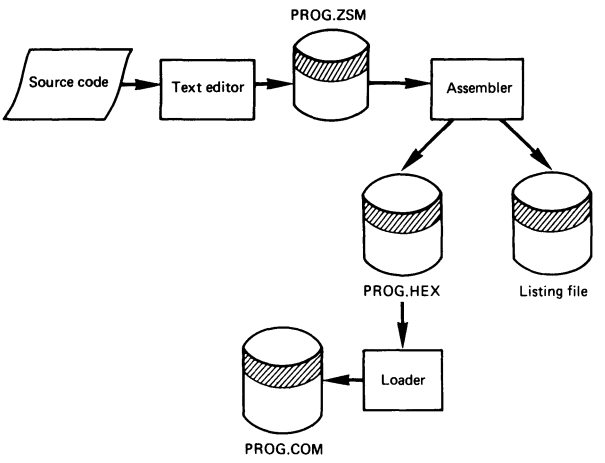


Figure 2.1 The assembly process

First of all, you need to create the program source file on disc using a text editor. If you are not familiar with a text editor you should learn to use one as soon as possible. The areas that you need to learn about first are

- How to input text
- How to delete text
- How to insert text
- How to alter text
- How to save the text on to disc and retrieve it

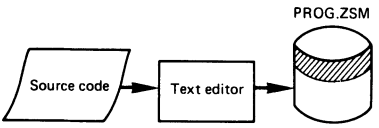
For the present you can type in the following program using PIP. Be very careful when you do this: if you make a mistake, start again. Now type

PIP PROG.ZSM=CON:

press RETURN and then input the following program source.

```
BDOS            =    0005H
FN_PRINT_STRING =    9
WARM_BOOT       =    0
                ORG  100H
                LD   SP,1000H
                LD   DE,MESSAGE
                LD   C,FN_PRINT_STRING
                CALL BDOS
                JP   WARM_BOOT
MESSAGE         DB   "Hello world!$"
END
```

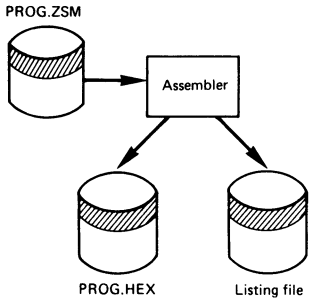
When you have finished doing this, type CTRL/Z. This will create the program source file PROG. ZSM.



You can now assemble your program using an assembler such as Research Machines *ZASM*. To do this, type

ZASM PROG

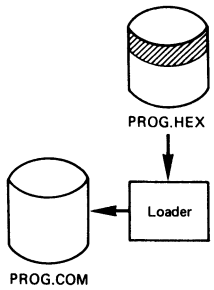
Your program source will be assembled into the file PROG.HEX and a listing will be output to the file PROG.PRN.



You can now produce a memory image of the program by typing

LOAD PROG

The LOAD utility is supplied with your CP/M system. It converts an intermediate form of the program into an actual executable memory image. LOAD reads in a .HEX file and creates a .COM file of the same name.



The memory image will be held in the file PROG.COM and it can be loaded and run by typing

PROG

When executed, the program will look like a CP/M command. This feature allows you to 'invent' new CP/M commands.

Building libraries

The simplest way of producing a library is to keep it in source form. Using this method, each library routine is kept in a separate file and individual routines can be merged with your program using a text editor.

Many assemblers will also let you merge source files of routines during assembly. They contain *INCLUDE* commands such as the following:

```
*I SCROP
```

where SCROP would be a file containing the source of a subroutine that handles screen output. By putting this command in the source of your main program, the source code in SCROP will be merged with the program when the program is assembled.

The previous two methods of building libraries have one major disadvantage: if you have a lot of programs which use one routine in particular and you modify that routine, you have to re-assemble all of the programs. To get around this problem, you can use a different type of library using *relocatable (REL) library files.* Each of these files contains a copy of a routine's code. When the main program has been assembled, you use a program called a *linker* to *link* the code of each routine into the program.

Each REL file also contains information which describes

- What will change if the routine's load address changes
- Which *names* are used in the routine

When you write your program, you can declare each name to be *global* (that is, it can be used by any routine) or *external* to the program itself. The linker looks at the names in each program instruction; if a name is declared as external, it searches the REL files that you specify to try to merge the appropriate routines.

The types of library we have covered up to now will be sufficient for most people. However, if you have a large program that contains, say, 200 routines, you might use a *librarian.* This is a program that takes the relocatable code from several files and merges them into one library file. As part of the operation, it marks each routine with its own identification so that a special linker can easily find the routine and link it into your programs.

To sum up on the subject of libraries you have a number of options, some of which cost nothing and some of which you may have to pay extra for. Whichever method you need depends upon the type of programs you write and what you can afford. However, you should use, at least, the most basic method in the interests of efficiency.

Testing and debugging

When testing a program, you should always assume that it will not work. You should put into it sufficient messages that tell you which parts of the program have been reached. When the program fails, you can then insert diagnostic code to tell you what is happening.

For the first few attempts you should use the simplest possible data as input. Only when the program works with that should you try anything more complex. You should next test the program using invalid data and, finally, the more complex, valid, data.

If your program has been well-designed and is modular in structure, testing and debugging it will be straightforward. You can print out registers at the beginning and end of the offending modules and work out what is happening in between. If the program has not been well-designed, testing and debugging can be very complicated.

As debugging is such an important part of programming and an art in itself, a separate chapter (chapter 7) has been devoted to it.

Optimisation

Optimisation can take three forms: making a program smaller, making it faster, and improving its user interface. Normally, the greatest impact in these areas takes place during the design phase and optimisation should not, therefore, be a major job.

You should optimise a program only when you are satisfied that it is working. To do so earlier is inefficient (you may decide during testing that a particular routine needs rewriting, or scrapping, and if it contains some optimised code, time would have been wasted).

If you are optimising a program for size, you should look at the largest modules first because that is where the benefits are likely to be greatest. Try to find areas of code that are the same, or similar, and turn them into routines. This can be useful even at the smaller scale if you are short of space. For example, if your program contains 20 calls to routines A and B and they always occur together, combining them into one call to C, which itself calls A and then B, will give code that is slightly slower but will save you about 60 bytes.

Summary

- Program development can be split into the following stages

  Design
  Program documentation
  Coding
  Assembling, loading and running
  Testing and debugging
  Optimisation

- Good design can reduce the overall development time
- Document the program as you are developing it. This helps you consolidate your idea of how the program works and makes debugging easier
- Optimisation is performed during the design stage and after coding. The most effective contribution is to select the most effective algorithm at the design stage. You should not try to optimise during the coding stage; instead, concentrate on writing clear code
- Don't re-invent the wheel. Use library routines where you can

3 Design

The sooner you start coding your program, the longer it is going to take.
It's like building a house with nails, hammer, bricks and cement,
but no blueprint: CHAOS!

Henry Ledgard

The design process is not and should not be a rigid one. There is scope for innovation, particularly when you are dealing with a completely new concept. However, a certain amount of self-discipline is necessary if you want a project to run smoothly.

Don't jump in at the deep end! You should always spend a significant amount of time on design before you start coding. In doing this you will reduce the testing and debugging processes which can easily take up half the program development time. There is no harm in writing 'test bed' code during the design process to explore the potential solutions of problems, but you should not be afraid to scrap the code and rewrite it from scratch once you have learned what you wanted from it.

Information gathering

The early part of the design process is mainly concerned with information gathering. Think about the application in general terms. What type of people will be using it and why are they using it? Do they need any special considerations? If they are unfamiliar with computers, be prepared to put in a lot of work on the user interface. Humans find it difficult to handle two new concepts simultaneously, and if computer is a completely new concept to users you have to take this into account.

Get a picture of what you need to do and research the subject if necessary. Analyse any problem areas by asking yourself

- What is the problem?
- What is the cause of the problem?
- Can the problem be broken into a number of parts, each of which could be solved easily?

The last of these three points is the most potent method of making life easier for yourself: divide and conquer!

As a result of the information gathering process, itemise any special conditions and the actions that you are going to take in relation to them. Tick them off when they have been dealt with.

The main design activity

General points

When you have gathered the information that you need, you should then work out a design solution in general terms. In particular, think out the main data structures and how they can be linked together in the form of tables.

For example, suppose you want to write a program that outputs the same data to a number of different screens at different times. What you could do is put details of each device (for example, screen width and number of lines) into a table as shown in figure 3.1. You can then have a generalised piece of code which handles output and which picks up from the table details of the appropriate screen before writing to it.

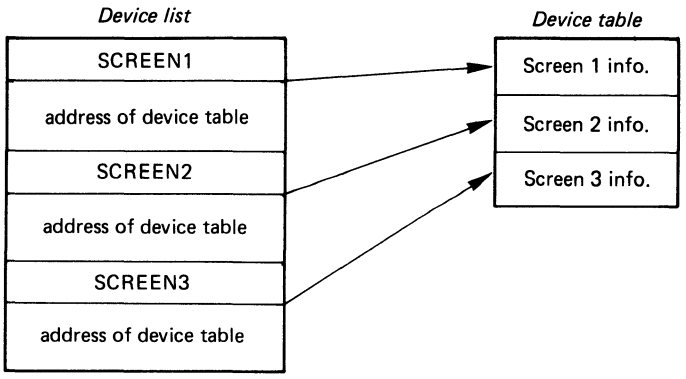


Figure 3.1 The use of tables

Programs with structures like this are described as being 'table-driven' and they are easy to maintain. To add a new device you merely make up a new device table, add a new entry to the device list and update the end of list pointer.

As well as looking at data structures you should organise your program so that it has a good structure. Make the main part of the program a series of calls to modules and put this in a prominent place. In designing the modules that will

make up your program, try to make them as general as possible, so that they can be used in different parts of the program and, indeed, in other programs that you write. Have you designed modules for other programs which could be used here as 'building blocks'? This type of approach will save you a lot of time over the years. In particular, it is easier to debug several small modules than one massive chunk of code; and if you have tested them for use in other programs, that's half the battle!

You should identify the key modules of your program and how they interact. Programming bugs often occur at interfaces so you should always pay particular attention to these and define them clearly.

Where there is a conditional statement (IF . . . THEN . . . ELSE) there is often a bug lurking! So check each of the branches in your program.

When dealing with numbers, decide what level of accuracy you need and try not to make your programs too sensitive. We know of one program that gave different results on two related computers because the floating point arithmetic units of each gave results to a slightly different degree of accuracy and the program used the full floating point accuracy.

One area that ought to be considered throughout the development process, but often is not, is validation. There is nothing worse than having your program fail at the most important moment. More often than not, other people will be present when it happens and they will not be impressed! You should therefore design tests and test data as part of the main design process.

Before you start to design tests, identify the *boundary conditions* of your program: in other words, where are its limitations? For example, suppose your program needs to request a file name from the keyboard. What happens if you type in an invalid filename? What happens if the file does not exist or is of the wrong type (a .COM file instead of, say, .TXT)? What happens if you specify an invalid drive name or a disc is not in the drive?

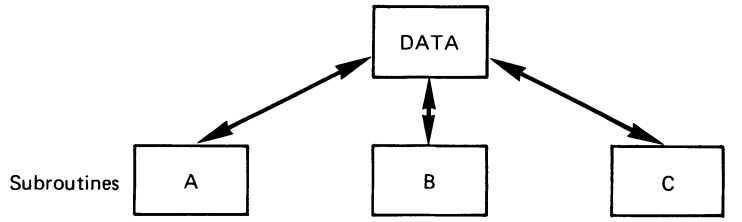
If your program opens an output file, what happens if the file already exists, the disc is full or becomes full before you finish writing the file?

Moving away from files, how well does your program validate the user input? Will it crash if you give it the wrong type of data (for example, text when it expects a number)? What happens when you just type a carriage return?

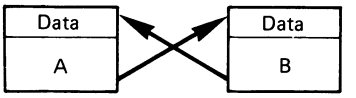
When you design your tests, bear all these things in mind and look for other boundary conditions. Test the areas beyond the boundaries, but do not forget to test valid data as well, sampling it at random if possible.

Subroutine coupling

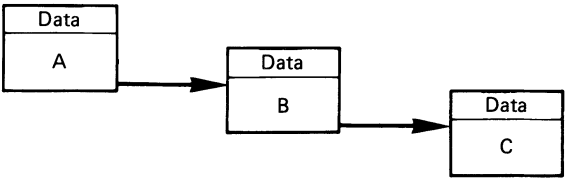
When subroutines share common data they are described as being *tightly coupled.* This type of arrangement is shown in the diagram below.



They may even be very tightly coupled, as shown in the next diagram. In this case, subroutines A and B access each other's data space.



Subroutines of the last type are very efficient in terms of machine resources, but they are also difficult to debug as there is a lot of interaction. A much easier system to debug is the *loosely coupled* one shown below.



Here, each subroutine has its own, local, data which no other subroutine can access and the interfaces between each are clearly defined.

In practice, you will probably use a combination of the loosely coupled and tightly coupled systems: each subroutine may have its own local data but also have access to a certain amount of 'global' data. However, whichever method you use, you should be aware of its advantages and disadvantages.

Program portability

There are two aspects to this: you might want to write software that can run on other computers, or you might want to take software from another computer to your own.

The most important thing to remember is to use only published interfaces: BDOS functions and BIOS calls (described in the Digital Research documentation). You should use BDOS functions rather than BIOS calls, if possible. However, even BIOS calls are better than talking to the firmware and hardware directly. You should also try to restrict your programming to the common set of BDOS functions; these are shown in appendix E.

The first thing to check when converting software to run on your computer is whether or not your TPA is large enough to hold the program. Most CP/M systems have 32K of TPA available but your source machine may have more.

Does the program need special devices or hardware that may not work on your computer and does it contain any machine-specific subroutines?

More points that may cause problems when using programs from other machines are listed below.

- Special function keys may have been used
- The way in which the source machine handles colour will probably not match the way in which your computer handles it
- The program may use screen attributes and control sequences. These are usually machine-dependent
- The program may make use of special areas of memory, for example, memory in page zero, or memory within the operating system and firmware
- The program may make assumptions regarding the speed of the target machine and its discs in matters concerning timing (especially in the case of real-time programs)
- The program may be designed for a printer with a special character set
- The program may use a non-standard character set
- The disc space on your computer may not be sufficient for the program. The number of drives and naming of drives may also be different

When writing portable software, many of these details can be set up in the form of a table and will thus be easier to change. In particular, it is best to put screen control sequences in a table and address each (variable-length) sequence using a separate table of pointers; this is then easy to change.

You should also isolate BDOS calls, or indeed, any input/output calls, to individual subroutines. This gives an overhead but ensures that the program is more suitable for transfer to non-CP/M systems.

When writing portable software you should take care not to use memory in page zero or in the operating system and firmware. In particular, you should not use the RST addresses; these may be used by the firmware.

There is always a trade-off between portability and performance when writing software. Sometimes it is more efficient to break the rules and take advantage of the special features of a machine. However, you should always remember that *you* might have to modify your program to make it run on a different machine.

The final stages of design

Before you go on to the process of coding your design you should

- Check out the design. Play devil's advocate and try to break your design
- Check and recheck your work

- Document your design, particularly if you are passing parts on for other people to code. It will also help if you need to modify the system later
- Don't write proper code (as opposed to 'test bed' code) during the design phase. Concentrate on functionality and inter-module interfaces

Summary

- Don't jump in at the deep end! Resist the temptation to start coding
- Write 'test bed' code, if necessary, but don't be afraid to scrap it
- Gather whatever information you need
- Think about the people who will be using the program. Put in a lot of work on the user interface, especially if the program is aimed at beginners to computing
- Analyse any problem areas:

  What is the problem?
  What is the cause of the problem?
  Can the problem be broken into a number of parts, each of which could be solved easily?

- Think about the main data structures. Table-driven structures are worth considering as they are easy to maintain
- Structure your program well and divide it into modules that could be used elsewhere
- Don't re-invent the wheel!
- Pay particular attention to module interfaces and conditional statements (IF . . . THEN . . . ELSE) as bugs are often found there
- Consider validation (testing) and design your test data
- Keep your subroutines as loosely coupled as possible (try not to let them access one another's data space)
- Decide whether or not your program may be put on to a different computer. If the answer is 'yes', design the program to be portable
- Before coding

  Check out the design
  Recheck it
  Document it

4 Using Simple BDOS Functions

BDOS function call	*Function code*	*Read Console Buffer*
Console Input	*Get Console Status*	*Return Version Number*
Console Output	*List Output*	*System Reset*
Direct Console I/O	*Print String*	

This chapter introduces you to the mechanism by which your programs communicate with CP/M: the *BDOS function call.* The first section describes the basic principles involved: the rest explain how to use the simple BDOS functions.

The BDOS function call

CP/M interfaces your program to the hardware and to access its facilities you have to make a request to the operating system. We will look at the way it does this by using a simple analogy.

Suppose that you have a friend living some distance from you and that you want him to do something for you. You might ask him to do it by putting your request in a letter and sending it to him. Similarly, with CP/M, you tell the BDOS what you want it to do by loading the C register with a number (the *BDOS function code*) that describes the request. You then send this to the operating system by calling the BDOS entry point at address 0005H.

As a simple example, suppose that you want to exit from your program and reset the system so that it is in a 'clean' state. You can do this as follows

```
BDOS            =   0005H
FN_SYSTEM_RESET =   0
                ...
                ...
                ...
EXIT:           LD  C,FN_SYSTEM_RESET
                CALL BDOS
```

Here, the function code 0 is used to tell CP/M that you want to perform a *system reset.* The piece of code loads function code 0 into register C and then calls the BDOS to perform the reset. Notice how we use the symbol

'FN_SYSTEM_RESET' to load register C and not the value 0. This makes the program easier to read and understand.

Going back to the analogy, suppose that you want your friend to send you his telephone number. Again, you go through the same procedure of putting your request in a letter and sending it off. However, this time you get something back: a telephone number.

CP/M has a number of BDOS functions that give you something back. One of them is *Return Version Number* (function 12), which returns two numbers in the registers H and L; an example of its use follows.

```
BDOS            =   0005H
FN_SYSTEM_RESET =   0
FN_VERSION_NO   =   12
                ...
                ...
                ...
                LD   C,FN_VERSION_NO
                CALL BDOS

                LD   A,H        ;Check system
                OR   A          ;type & version.
                JR   Z,PROCESS  ;Jump if CP/M

EXIT:           LD   C,FN_SYSTEM_RESET
                CALL BDOS
```

Here, again, you go through the procedure of loading the C register with the function number (this time, the value 12) and then calling the BDOS entry point. However, when control passes back to your program, CP/M will have inserted in registers H and L two numbers that describe which type and version of the operating system your program is running under. The number in H describes the system type.

H	*System type*
0	CP/M
1	MP/M
2	CP/NET

The number in L describes the version number and the full range is shown on page 140.

In this example, we check the system type and do a system reset if the system is not CP/M. The normal mode of use of function 12 would be where we want to write a program that can run under different operating systems in the CP/M family. This program can be made to run under all versions, but we can improve it

by making use of the special facilities of each version. To do this, however, we need to know the version under which we are running, and Return Version Number tells us this.

Before we go on to look at more of the simple BDOS function calls there are two other important points that you need to remember. First of all, you need to define your own stack; if you don't, you may overwrite CP/M. In short test routines you may not have this problem but in larger programs you will.

The other point to remember is this: CP/M makes extensive use of the Z80 registers but does not save them before trampling them into the dust. You should thus save any registers you want preserving by 'pushing' them on to the stack or saving them in fixed memory locations before any BDOS call and 'popping' them on return. Remember that *the BDOS corrupts 99 per cent of household registers!*

Simple output using BDOS functions

CP/M allows you to send ASCII characters to the screen and printer using BDOS function calls. This section describes how the following three are used.

Function code	*Function*
2	Console output (sends one ASCII character to the screen)
5	List output (sends one ASCII character to the printer)
9	Print string (sends a string of ASCII characters to the screen)

With these and some other function calls, you need to pass information other than the function code to the operating system. For example, with *Console Output* (function 2), you put the character you want to output in *register E* before you call the BDOS.

```
BDOS            =   0005H
FN_CONSOLE_OUT  =   2
                ...
                ...
                ...
                LD  E,'Q'
                LD  C,FN_CONSOLE_OUT
                CALL BDOS
                ...
                ...
```

If you want to output more than one character to the screen, you can use *Print String* (function 9). With this, you need to give CP/M two pieces of information

- The *address of the start of the string*
- The *position of the last character* in it

The start address of the string is a 16-bit quantity, so we use the 16-bit register pair DE to hold it. To define the end of the string, we use CP/M's special end-of-line character $.

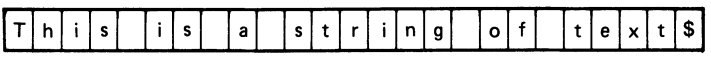
The following piece of code shows how you can output a string of text using Print String (function 9).

```
BDOS            =   0005H
FN_PRINT_STRING =   9
END_OF_STRING   =   '$'

MESSAGE:        DEFM 'This is a string of text'
                DEFB END_OF_STRING
                ...
                ...
                ...
                LD   DE,MESSAGE    ;Pick up message
                                   ;start address.

                LD   C,FN_PRINT_STRING
                CALL BDOS          ;Print message.
                ...
                ...
```

The string of text is stored in successive bytes, (where the last byte is a $ character) as shown below. This can be considered to be a one-dimensional array.



You could have put the $ at the end of the string, instead of using 'END_OF_STRING'. We have written the code in the way shown above for ease of maintenance: if the terminator character were to change, program modifications can be minimised.

A consequence of the use of $ as a terminator is this: you cannot print a $ character using Print String. You must, instead, use Console Out (function 2).

The last BDOS function that we will look at in this section is *List Output* (function 5). This allows you to send *one* ASCII character to the printer and it is similar in use to Console Output (function 2): you load the ASCII character that you want to print into register E, load the function code 5 into register C and then call the BDOS. The character is *not* copied ('echoed') to the screen.

In the examples that we have shown in this section, only alphabetic ASCII characters have been shown. You can, if you wish, use output control characters as well, and the full range is shown in appendix C. Note that the effects of some of

them differ between systems; you should thus check your particular CP/M implementation and update the table in appendix C accordingly.

Simple input using BDOS functions

As well as being able to send information to the screen, your program may want to accept information from the keyboard. CP/M allows you to do this using the following two BDOS function calls.

Function code	*Function*
1	Console Input (accepts one ASCII character from the keyboard)
10	Read Console Buffer (accepts a number of ASCII characters from the keyboard)

The rest of this section describes how they are used.

The simplest way of reading information from the keyboard is by using *Console Input* (function 1). This suspends your program until a key on the keyboard has been pressed and then returns control to your program with the key character in the A register. This character is also echoed to the screen if it is a printable character (see the description of Console Input in appendix D for full details).

The following subroutine shows how this function is used.

```
BDOS              =    0005H
SPACE             =    20H
FN_CONSOLE_IN     =    1
                  ...
                  ...
                  ...
WAIT_FOR_START_KEY:
                  LD   C,FN_CONSOLE_IN
                  CALL BDOS

                  CP   SPACE           ;Is it a space?
                  JR   NZ,WAIT_FOR_START_KEY

START_KEY_PRESSED:
                  RET
```

This subroutine waits for a key to be pressed on the keyboard. When a key is pressed, the BDOS call returns with the key character in the A register. The code then tests to see if the key pressed was a space. If it was, control returns to the

calling routine; otherwise, the code waits for another key to be pressed.

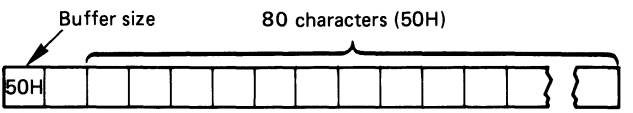
Sometimes it is useful to be able to read a single character from the keyboard, but more often than not you will want to read a number of characters into a buffer. You can do this using *Read Console Buffer* (function 10). With this, you tell CP/M where the buffer is in your program, and how big it is, by

- Putting the buffer start address into the register pair DE
- Putting the number of bytes *available* (1 to 255) into the first byte of the buffer

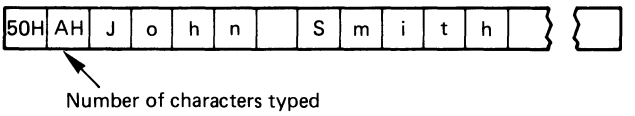
The following piece of code shows how to use Read Console Buffer.

```
BDOS            =    0005H
FN_READ_BUFFER  =    10
                ...
                ...
BUFFER_LENGTH   =    80
BUFFER:         DEFB BUFFER_LENGTH
                DEFS BUFFER_LENGTH+1
                ...
                ...
                LD   DE,BUFFER      ;Set up buffer
                                    ;start address.
                LD   C,FN_READ_BUFFER
                CALL BDOS
                ...
                ...
```

This code will set up the buffer as follows:



If you were then to type in the text 'John Smith' and press RETURN, the buffer would now look like this:



Any text typed in up to the point where you pressed RETURN is always stored from the third byte in the buffer onwards. The number of characters typed

in is stored in the second byte. If you forget to reserve storage for the buffer, CP/M will clobber your program by overwriting it! *You have been warned!*

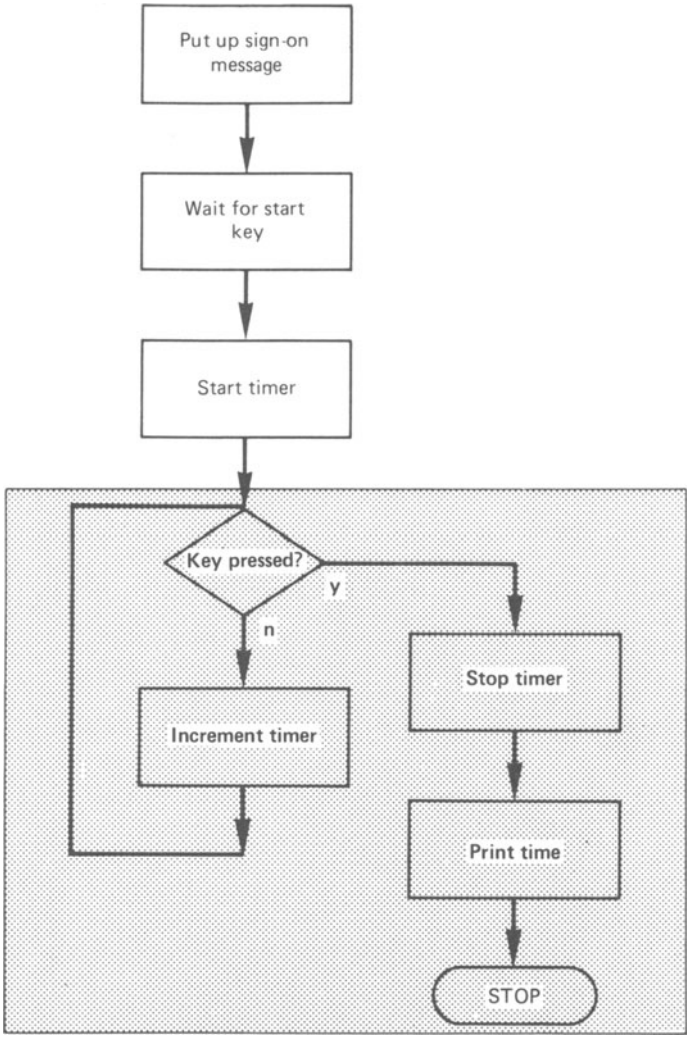
A particular benefit of 'Read Console Buffer' is its simple editing facilities. These allow you to backspace a character and delete a character, for instance. Full details of the editing facilities are given in appendix D under the description of *Read Console Buffer.*

A more efficient mode of input

In some situations you may want to test if a character has been input at the keyboard and continue processing if it has not. You would need to do this, for example, in an arcade-type games program, or a program that tests your reaction time.

In this type of situation you could use *Get Console Status* (function 11). This tests if a key has been pressed at the keyboard and returns the value 1 in register A if it has; otherwise, it returns the value zero. When you know that a key has been pressed, you can get its value by using either Console Input (function 1) or Read Console Buffer (function 10).

Suppose that you want to write a program that tests your reaction time. The following flowchart shows what is needed.


Put up sign-on message
Wait for start key
Start timer
Key pressed?
y
n
Stop timer
Increment timer
Print time
STOP

The part shown stippled is the most significant from our point of view. The code for it might look like that shown below.

```
TIMER_LOOP:       LD   C,FN_CONSOLE_STATUS  ;Has key been pressed?
                  CALL BDOS

                  OR   A                    ;Set flags:
                                            ;  - Z  no key pressed
                                            ;  - NZ key pressed.
                  JR   NZ,STOP_THE_CLOCK
                  CALL INCREMENT_THE_CLOCK
                  JR   TIMER_LOOP

STOP_THE_CLOCK:   CALL STOP_TIMER
                  CALL PRINT_TIME
                  CALL STOP_PROGRAM
```

Note how Get Console Status (function 11) has been used here to break out of the timer loop when the key has been pressed. The complete reaction tester program is shown in appendix A.

Bypassing the BDOS

The input/output functions described up to now have a number of restrictions. In particular, certain control characters (CTRL/C, CTRL/S, CTRL/Q and CTRL/P) have special meanings when typed at the keyboard. In the case of CTRL/C this might reboot the operating system! If you input CTRL/S this stops output to the screen, and if you input TAB characters these will be expanded.

What actually happens is shown in figure 4.1. Part of the BDOS analyses which function call you have requested. We will call it the *function analyser.* This then decides which routine in the BDOS it will call to handle your request. In the case of most console operations, control passes through the appropriate console-handling routine in the BDOS and this in turn calls a BIOS console routine.

When control passes through the BDOS console-handling routines, certain actions take place. On input, they check for special characters such as CTRL/C, CTRL/S, etc. On output, they expand tab characters.

Under certain circumstances, it is useful to be able to bypass this processing of special characters by the BDOS console-handling routines. *Direct Console I/O* (function 6) allows you to do this, as shown in figure 4.2. Here, you can assume that control passes directly from the BDOS function analyser to the BIOS console-handling routines. As a consequence, you can get 'raw' data into and out of the system.

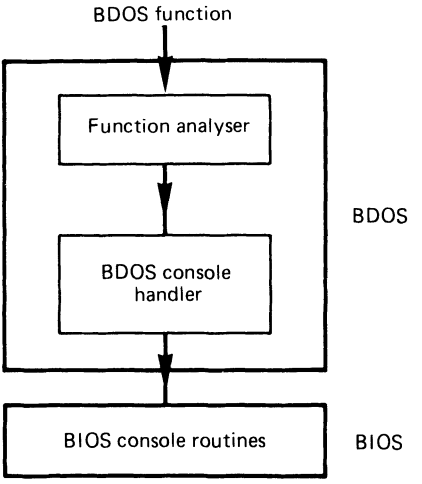


Figure 4.1 Flow of control in a normal BDOS call

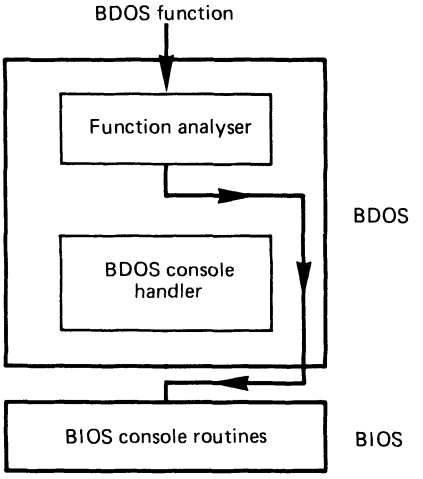


Figure 4.2 Flow of control using direct console I/O

Note that you must not mix Direct Console I/O and the other input/output functions described in this chapter; if some functions go through the BDOS and others bypass it the results can be unpredictable.

To save you needing to mix them, Direct Console I/O provides you with most of the facilities already described. You can use it to input or output a character.

You can also use it to check the keyboard status, so you could still implement programs like the reaction timer shown in the previous section.

The following piece of code outputs the character *Y* to the screen using Direct Console I/O.

```
BDOS            =   00005H
FN_DIRECT_IO    =   6
                ...
                ...
                ...
                LD    E,'Y'         ;Load character to
                                    ;be output.
                LD    C,FN_DIRECT_IO
                CALL  BDOS
                ...
                ...
```

Here, the character to be output is first loaded into the E register, just as in Console Output (function 2).

When you are using Direct Console I/O for input it does not wait for a key to be pressed: it gives you either the character input, if one is there, or zero. Thus, you need to check that a character has been transferred (by testing for a non-zero value) each time that you use it. A side effect of this is the fact that Direct Console I/O cannot be used to input NULL characters (CTRL/@) as these are zero.

The following code shows how Direct Console I/O can be used to input the character CTRL/C (3) without rebooting CP/M. You first load register E with the value OFFH to show that you want to *input* a character. On return from the BDOS call, any character transferred is in register A.

```
BDOS            =   0005H
FN_DIRECT_IO    =   6
CTRL_C          =   3
INPUT_MODE      =   OFFH
                ...
                ...
                ...
WAIT_FOR_CHAR:  LD    E,INPUT_MODE      ;Indicate input request.
                LD    C,FN_DIRECT_IO
                CALL  BDOS
                JR    Z,WAIT_FOR_CHAR   ;Character input?

                CP    CTRL_C            ;Test for CTRL/C.
                JP    Z,EXIT
PROCESS_CHAR:   ...
                ...
EXIT:           ...
                ...
```

Notice how the keyboard status is checked to see if a character has been input.

Another useful feature of Direct Console I/O is the fact that when you use it for input, text is not echoed to the screen. You could, thus, use it for password detection, as in the example shown below. Here, the subroutine READ_PASSWORD_ECHO_STARS allows you to type in a password and store it in a buffer so that you can process it outside the subroutine. Each time that you type a character of the password, the subroutine will print a star (*) character so that no-one can watch you typing the password. On entry to the subroutine you should put the buffer address in registers HL and the number of bytes it contains in register B.

```
BDOS            =   0005H
FN_DIRECT_IO    =   6
INPUT_MODE      =   0FFH

READ_PASSWORD_ECHO_STARS:

Read_password_loop:

                CALL READ_A_CHARACTER
                LD   (HL),A             ;Save it in buffer.

                LD   A,'*               ;Print a star.
                CALL DISPLAY_A_CHARACTER

                INC  HL                 ;Next character position.
                DJNZ Read_password_loop ;Repeat until finished.
                RET
    ;.....................................................

READ_A_CHARACTER:
                PUSH HL                  ;Save registers used.
                PUSH BC

Wait_for_key:   LD   E,INPUT_MODE
                LD   C,FN_DIRECT_IO
                CALL BDOS

                OR   A                   ;Key pressed?
                JR   Z,Wait_for_key      ;No!

Key_pressed:    POP  BC                  ;Restore registers.
                POP  HL
                RET
    ;.....................................................
```

```
DISPLAY_A_CHARACTER:
                PUSH HL                 ;Save registers used.
                PUSH BC

                LD   E,A                ;Get character to output.
                LD   C,FN_DIRECT_IO
                CALL BDOS

                POP  BC                 ;Restore registers
                POP  HL
                RET
    ;.....................................................
```

If you have the choice of using Direct Console I/O or the other input/output functions, it is better to avoid Direct Console I/O. You can input and output only one character at a time with it. Also, the line editing facilities that are available with Read Console Buffer cannot be used with Direct Console I/O.

To conclude this section, we must stress one thing yet again: *you should not mix Direct Console I/O with the other console input/output functions.*

Summary

- The *BDOS function calls* allow you to access operating system facilities. The calls are summarised on page 109 and each one is described in detail in appendix D. To use one of them you load the *function code* which identifies the facility into register C, and then call the BDOS at address 0005H
- In some cases you pass information to the BDOS via registers D and E. The BDOS may also pass information back to you via one or more of the A, H and L registers. The flow of information is shown in table 4.1
- *Note that you must not mix Direct Console I/O and the other console input/output functions described in this chapter. If some functions go through the BDOS and others bypass it the results can be unpredictable.*

  To save you needing to mix them, Direct Console I/O provides you with most of the facilities already described. You can use it to input or output a character. You can also use it to check the keyboard status. However, you cannot use it to read or output a string of characters in one call
- Define your own stack, otherwise you might overwrite CP/M's stack
- Remember that *the BDOS corrupts 99 per cent of household registers!*

Table 4.1 Flow of information in BDOS calls

Function	Fn Code	Information passed to BDOS	Information returned by BDOS
System Reset	0	None	None
Console Input	1	None	ASCII character in A
Console Output	2	ASCII character in E	None
List Output	5	ASCII character in E	None
Direct Console I/O	6	ASCII character in E (output). OFFH in E (input)	None ASCII character in A, or 0 if key not pressed
Print String	9	Address of start of string in DE. $ character as terminator at end of string	None
Read Console Buffer	10	Buffer start address in DE. Number of bytes available in first byte of buffer	Number of bytes returned is in second byte of buffer. Bytes returned start at third byte of buffer
Get Console Status	11	None	Number in A tells you if key has been pressed 0 - no key pressed 1 - Key pressed
Return Version Number	12	None	System type (register H) System version (register L)

5 File Handling

Appending to a file	*Directory*	*Filename*	*Sector*
Binary file	*Disc*	*Filetype*	*Search for First Entry*
Blocking/De-blocking	*Disc data buffer*	*Open File*	*Search for Next Entry*
Close file	*DMA Address*	*Random access*	*Set DMA Address*
Command line tail	*Drive name*	*Read Random*	*Text file*
Create File	*Extent*	*Read Sequential*	*Write Random*
Default Data Buffer	*File*	*Record*	*Write Random with Zero Fill*
Delete File	*File Control Block (FCB)*	*Rename File*	*Write Sequential*

Discs are one of the most versatile ways of storing computer data. They are compact and can hold a large amount of information which can be retrieved quickly and easily. This chapter and the next describe the ways that you can access discs and the files into which they are divided. The present chapter describes file operations and the BDOS functions that they use. The next chapter covers more sophisticated disc operations and their BDOS functions.

Discs and files

Disc file storage is similar to the storage method used in libraries, so we will look at this first (see figure 5.1). Each book in a library contains *information* and it has an entry in a central index. This index is a type of *directory* which tells you where to look for books; it consists of a card for each book. On each card is the name of the book and its author, together with a number that tells you where the book is in the library.

You can think of magnetic discs as containing *files* which correspond to the books in a library. Each file can hold information such as text, programs and raw data. For each file, there is an entry in a *directory* which holds, for example, the *filename* and the position of the file on the disc.

Let us take this analogy a bit further. If you want to find a book in the library and read it, you look at the index (directory) to find out where the book is. Then you take the book from its place in the library, you *read* its contents and, finally, *close* it and put it back. With the equivalent disc filing system, when you want to

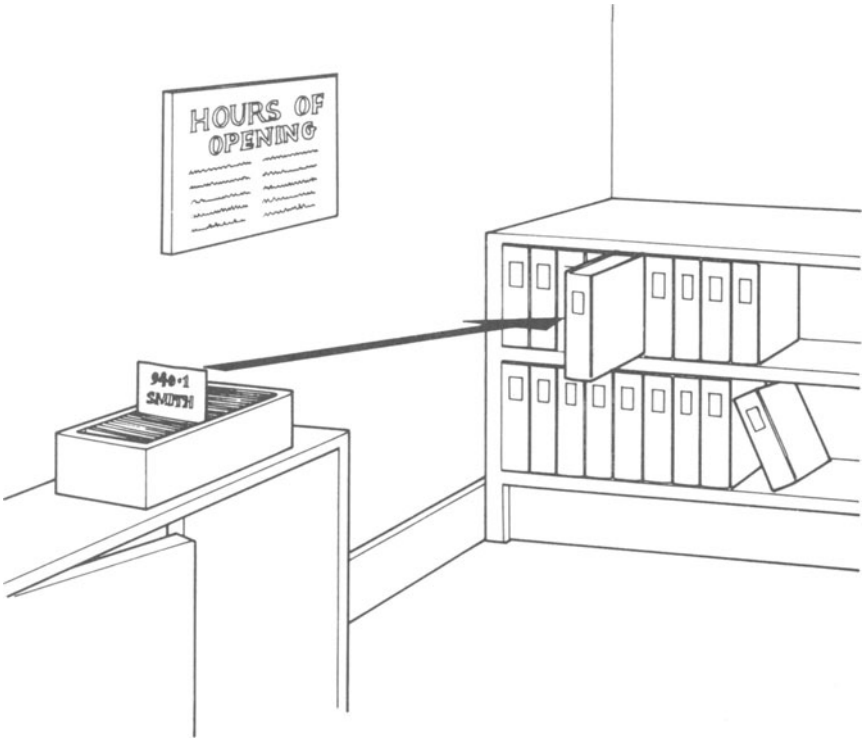


Figure 5.1 Library analogy

read a file, CP/M would look at the disc directory to see where the file is and *open* it for use by your program. Your program would then *read* information from the file and, finally, *close* it. The action of opening the file is comparable to looking in the library's book index.

Going back to the library analogy for the last time, when the librarian wants to put a new book into the library, he creates an entry for it in the index and then puts the book into its correct position in the library. With discs, the procedure is similar, but this time the actions are carried out by your program and CP/M. When your program asks CP/M to *create*, or make a new file, CP/M creates a new entry for the file in the disc's directory, specifying the filename. Your program can then *write* records to the file and, finally, *close* it.

The librarian can put a book in the library, but no-one else will know where it is until the library's index has been updated. Similarly with disc files, CP/M will not know where the file is on the disc until the file has been written and closed, at which point the directory is updated.

To summarise the points we can learn from this library analogy: a disc is organised into a number of files and these are indexed by a short directory. When your program creates, or makes, a file three things happen

- CP/M creates a new directory entry for the file
- Your program writes information to the file
- Finally, your program closes the file

When your program reads a file, again three things happen

- CP/M opens the file (it looks in the directory to find out where the file exists on the disc)
- Your program reads information from the file
- Finally, your program closes the file

Now let us look at how to create a disc file, and read from it, in more detail.

Creating a simple file

Look at the following program. It creates a disc file called DATAFILE.DAT, reads a line from the keyboard and writes it to the disc file. Finally, it closes the file. Each line is written to disc when you press RETURN and you can stop writing lines to disc by typing '*' then pressing RETURN.

```
TPA                     equ     100h
BDOS                    equ     5
default_data_buffer     equ     80h
default_drive           equ     0

null                    equ     0
clrscreen               equ     1FH
cr                      equ     0DH
lf                      equ     0AH
end_of_string           equ     '$'

max_line_length         equ     126.

        org     TPA

        LD      SP,my_top_stack
        CALL    display_signon_msg

        CALL    CREATE_DATA_FILE
        INC     A                       ; test error return for -1
        JR      Z,there_was_an_error
```

```
input_loop:
        CALL    READ_LINE               ; get line of text from kbd
        CALL    TEST_FINISHED           ; see if line was just a '*'
        JR      Z,LE_FINI

        CALL    WRITE_LINE_TO_FILE
        OR      A                       ; test if successful, A = 0.
        JR      NZ,there_was_an_error

        JR      input_loop              ; and round again

LE_FINI:
        CALL    CLOSE_DATA_FILE
        JP      EXIT

there_was_an_error:
        CALL    display_error_msg
        JP      EXIT

;------------------
datafile_fcb:

        DEFB    default_drive           ; currently selected drive
        DEFM    'DATAFILE'              ; name of file
        DEFM    'DAT'                   ; file type
        DEFB    0,0,0,0,0,0,0,0         ; pad the rest with zeros
        DEFB    0,0,0,0,0,0,0,0
        DEFB    0,0,0,0,0,0,0,0
```

We will now look at the subroutines that form this program. First, look at the subroutine CREATE_DATA_FILE; it is shown below.

```
FN_CREATE_FILE  equ     16h

CREATE_DATA_FILE:
        LD      DE,datafile_fcb         ; Ptr to FCB
        LD      C,FN_CREATE_FILE
        CALL    BDOS
        RET
```

The first BDOS function, *Create File* (function 22), creates an entry in the directory for the file DATAFILE.DAT. It works in the same way as the simple BDOS functions described in the previous chapter: you load register C with the function code and pass information to CP/M using the register pair DE. If CP/M is able to create an entry in the directory it will return a number between 0 and 3

in the A register, otherwise it will return −1. The program above shows you how to check this information.

The information needed by CP/M when it creates a file is the drive name, filename and file type. This is held in a 36-byte data area called a *File Control Block,* or *FCB.* When you use the Create File function, the register pair DE contains the address of this FCB. For the present, we shall not look at the contents of the FCB (they are described in detail later in this chapter).

Lines are read from the keyboard by the subroutine READ_LINE. The subroutine WRITE_LINE_TO_FILE shown below then uses the BDOS function *Write Sequential* (function 21) to write a line to the file.

```
FN_WRITE_SEQUENTIAL     equ     15h

WRITE_LINE_TO_FILE:
        LD      DE,datafile_fcb
        LD      C,FN_WRITE_SEQUENTIAL
        CALL    BDOS
        RET
```

With Write Sequential, CP/M needs two types of information

- The address of an FCB that describes the file
- The address of a 128-byte data area that contains the data to be written to the file. This is called the *disc data buffer*

You must use the same FCB here as was used in the Create File operation. You can pass its address to CP/M in the same way: via the DE register pair. If you want, you can create your own disc data buffer. However, CP/M allocates a *default data buffer* at the address 0080H and we will use this for the present. Now, all we need do is ensure that the data we want written is placed in the 128 byte block of memory starting at address 0080H. This is done by the subroutine READ_LINE.

Again, when the write operation is completed, CP/M returns a value in the A register to tell you whether or not it was successful: a zero value indicates a successful operation and any other value an unsuccessful one. The code in the program above shows you how to use this information.

The final operation that you perform when creating this simple file is to close the file using the BDOS function *Close File* (function 16). This is shown below in the subroutine CLOSE_DATA_FILE.

```
FN_CLOSE_FILE   equ     10h

CLOSE_DATA_FILE:
        LD      DE,datafile_fcb
        LD      C,FN_CLOSE_FILE
        CALL    BDOS
        RET
```

Again, you pass the address of the FCB to CP/M via register pair DE. If the operation is successful, register A will hold a number from 0 to 3. If it is unsuccessful, the register will hold the value −1.

Reading a simple file

If you now want to read data back from the file DATAFILE.DAT and print it on the screen, you can do this with the program shown below.

```
TPA                     equ     100h
BDOS                    equ     5
default_data_buffer     equ     80h
default_drive           equ     0
null                    equ     0
clrscreen               equ     1FH
cr                      equ     0DH
lf                      equ     0AH
end_of_string           equ     '$'
max_line_length         equ     126.

        org     TPA
        LD      SP,my_top_stack
        CALL    display_signon_msg

        CALL    OPEN_DATA_FILE
        INC     A                       ; test error return for -1
        JR      Z,there_was_an_error

display_loop:
        CALL    READ_LINE_FROM_FILE     ; get line of text from file
        JR      NZ,LE_FINI              ; test for end of file

        CALL    WRITE_LINE              ; go and print it
        JR      display_loop            ; and round again
```

```
LE_FINI:
        CALL    CLOSE_DATA_FILE
        JP      EXIT

;-----------------

there_was_an_error:
        CALL    display_error_msg
        JP      EXIT

;-----------------
datafile_fcb:

        DEFB    default_drive           ; currently selected drive
        DEFM    'DATAFILE'              ; name of file
        DEFM    'DAT'                   ; file type
        DEFB    0,0,0,0,0,0,0,0         ; pad the rest with zeros
        DEFB    0,0,0,0,0,0,0,0
        DEFB    0,0,0,0,0,0,0,0

;----------------
```

The first subroutine of this program, OPEN_DATA_FILE, is shown below.

```
FN_OPEN_FILE    equ     0FH

OPEN_DATA_FILE:
        LD      DE,datafile_fcb         ; Ptr to FCB
        LD      C,FN_OPEN_FILE
        CALL    BDOS
        RET
```

In OPEN_DATA_FILE, the BDOS function *Open File* (function 15) is similar in concept to Create File, but it operates only on existing files. Again, the address of the FCB is passed to CP/M using the register pair DE. If the operation was successful, CP/M returns a number between 0 and 3 in the A register. If it was unsuccessful, the register will hold the value −1.

The program now reads data from the file via the subroutine READ_LINE_FROM_FILE, which is shown below.

```
FN_READ_SEQUENTIAL      equ     14H

READ_LINE_FROM_FILE:
        LD      DE,datafile_fcb
        LD      C,FN_READ_SEQUENTIAL
        CALL    BDOS
        OR      A                       ; set flags
        RET
```

In this subroutine, data is read using the BDOS function *Read Sequential* (function 20). The lines of data are read back in sequential order. Again, CP/M needs the address of an FCB to tell it which file to read, and a data area into which it can read the data. As before, you pass the FCB address to CP/M via the register pair DE and CP/M uses the default data buffer at address 0080H to hold the data read from disc.

CP/M tells you if the read operation has been successful by putting a zero value in the A register; any other value tells you that it was unsuccessful (for example, the end of file may have been reached).

As each line of data is read from disc, it is printed on the screen by the subroutine WRITE_LINE.

Whenever you read or write a file, the last operation you should perform is to close the file. This is done in the subroutine CLOSE_DATA_FILE which is the same as the routine used in the previous section, when the file was written.

Adding data to the end of a file

There are not many applications where you would write a file once and not need to change it at some later date. You might want to add further data to the end of the file, or change some of the data you have already written. For the time being we shall ignore the latter type of action (it is covered later in this chapter) and concentrate on the former.

If you want to add data to the end of your file you can do this easily by

- Opening the file
- Reading up to the end of the file
- Writing the new data using Write Sequential

The following program writes one new line to the end of your file DATAFILE.DAT.

```
TPA                     equ     100h
BDOS                    equ     5
default_data_buffer     equ     80h
default_drive           equ     0

null                    equ     0
clrscreen               equ     1FH
cr                      equ     0DH
lf                      equ     0AH
end_of_string           equ     '$'

max_line_length         equ     126.
```

```
        org     TPA
START:
        LD      SP,my_top_stack
        CALL    display_signon_msg

        CALL    OPEN_DATA_FILE
        INC     A                       ; test error return for -1
        JR      Z,there_was_an_error

read_to_end_of_file:
        CALL    READ_LINE_FROM_FILE     ; get line of text from file
        JR      Z,read_to_end_of_file   ; loop until end of file

input_loop:
        CALL    READ_LINE               ; get line of text from kbd
        CALL    TEST_FINISHED           ; see if line was just a '*'
        JR      Z,LE_FINI

        CALL    WRITE_LINE_TO_FILE
        OR      A                       ; test if successful, A = 0
        JR      NZ,there_was_an_error

        JR      input_loop              ; and round again

LE_FINI:
        CALL    CLOSE_DATA_FILE
        JP      EXIT

;------------------

datafile_fcb:

        DEFB    default_drive           ; currently selected drive
        DEFM    'DATAFILE'              ; name of file
        DEFM    'DAT'                   ; file type
        DEFB    0,0,0,0,0,0,0,0         ; pad the rest with zeros
        DEFB    0,0,0,0,0,0,0,0
        DEFB    0,0,0,0,0,0,0,0

;----------------

FN_SYSTEM_RESET equ     0

EXIT:
        LD      C,FN_SYSTEM_RESET
        CALL    BDOS                    ; exit to CP/M

        RET                             ; just in case
```

This is similar to the program used in the previous section, but this time when the end of file is detected the subroutine WRITE_LINE_TO_FILE is called. This is the same as the subroutine of the same name that was used to write the initial file.

As in the previous operations which created and read from the file, you must close the file when you have finished with it. This is done, again, by the subroutine CLOSE_DATA_FILE.

More about files

The previous two sections described how to create a simple file and read from it. This section looks at the contents of FCBs, files and data areas in more detail.

FCBs and their contents

As we discovered in the first two sections of this chapter, your program uses a *File Control Block (FCB)* to pass the name of the file (or filename) to CP/M, and CP/M stores pointers to the information on disc in it. You must set up an FCB yourself for each file accessed by your program, and pass its address to CP/M in the register pair DE.

The FCB consists of a 33-byte block of memory for sequential access and a 36-byte block for the random access methods described later in this chapter. To avoid mistakes, we suggest that you always reserve 36 bytes for every FCB.

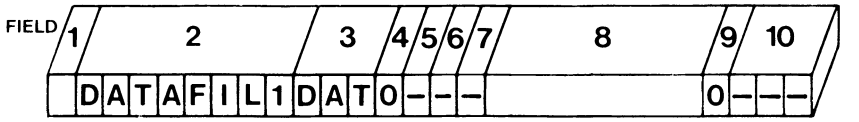
The format of an FCB is shown in figure 5.2. When you create or open a file it is your responsibility to complete the lower 13 bytes of your FCB (fields 1 to 4) and *set the remaining fields to zero.*

When you create a file, the contents of its FCB are stored in the disc directory. The FCB is updated as you write records to the file. If, when you close the file, records have been written to it, information in the FCB is written to the directory. When you subsequently open the file again, this information is read back from the directory and used by CP/M.

There is one final, important, point to remember about FCBs. *After you have created or opened a file, you should not alter the contents of its FCB. If you do, this may confuse the system.*

Sectors and blocking

CP/M files can be regarded as sequences of 128-byte chunks which we will call *sectors.* These are numbered from sector 0, and a file can contain any number of sectors up to the full capacity of the drive. In practice, the file is divided into a number of 16K byte segments called *extents,* and these extents could be spread across an entire disc with other files intervening. However, your programs will



Field	Bytes	Description
1	0	Drive code (0-16) 0 means use default drive for file 1 means drive A 2 means drive B ... 16 means use drive P
2	1-8	Contain the file name in ASCII upper case with high bit = 0
3	9-11	Contain the file type in ASCII upper case, with high bit = 0
4	12	Contains the current extent number, normally set to 00 by you, but in range 0-31 during file I/O
5	13	Reserved for internal system use
6	14	Reserved for internal system use, set to zero on call to OPEN, MAKE, SEARCH
7	15	Sector count for the extent in field 4; takes on values from 0 to 128
8	16-31	Filled-in by CP/M, reserved for system use
9	32	Current sector to read or write in a sequential file operation, initially set to zero by you
10	33-35	Optional random record pointer in the range 0-65535, with overflow to byte 35. Bytes 33 and 34 constitute a 16-bit value with low byte in byte 33 and high byte in byte 34

Figure 5.2 Format of an FCB

regard a file that consists of several such extents as being one continuous chunk of data.

If you want to write data as *records* of less than 128 bytes each, you can do this in two ways

- By padding out the data in each record to form a complete 128-byte sector
- By *blocking* a number of records into each sector when they are written and *de-blocking* sectors when they are read

In both these cases, your program must do the extra work itself: CP/M does not concern itself with the details of how 128-byte sectors are used.

Types of file

All files look the same to CP/M. However, from our point of view they can be divided into two groups

- Text files
- Binary files (for example, .COM files)

Text files are regarded as containing ASCII characters. One 128-byte sector can hold several lines of text and the last sector of a text file has its last 128 byte sector 'padded' with CTRL/Z (1AH) characters. Programs like WordStar use this type of file. Your program source files are other examples; in these, each 'line' of the file is denoted by a carriage return linefeed sequence (0DH followed by 0AH).

Binary files include .COM files and data files, and they contain what you can regard as a string of binary data. Wherever the number (1AH) occurs in these files it does not represent end of file (CTRL/Z). Instead, whenever the end of file condition occurs, CP/M returns the appropriate value in register A after the operation you performed.

Data areas

Now let us look at the other data area used by the BDOS file handling functions: the *disc data buffer.* This is used to hold records that you want to write to a file and those read from it. The start address of this buffer is called the *Direct Memory Access (DMA) Address.* This name is a throwback to the earlier days of CP/M and is not a very good one. However, we will retain its use here.

A default buffer is provided at address 0080H, but you can allocate your own buffer if you want using the BDOS function *Set DMA Address* (function 26). In most cases you would probably want to do this. You might, for example, want to use a number of files simultaneously in the same program (and each normally requires its own data area).

Another situation where you might want to use Set DMA Address is when you want to write several sectors of data to the disc one after the other (see figure 5.3). Here, it would be best if you organise your 128-byte sectors of data sequentially in memory. You can then change the data buffer address to the start of each chunk of memory (using Set DMA Address) before each Write Sequential operation.

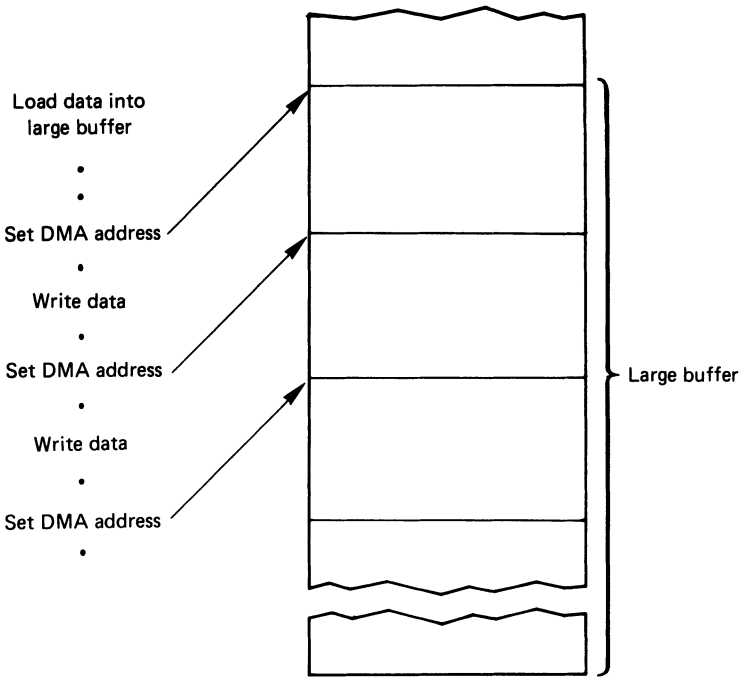


Figure 5.3 Handling large chunks of data

The complete sequence of operations is shown by the following piece of code.

```
sector_length           equ     128.
sector_count            equ     buffer_length/128

        LD      B,sector_count
        LD      DE,sector_length
        LD      HL,buffer       ; buffer is a large contiguous area
                                ; of data we want to write to file
write_buffer:
        push    HL              ; save registers we use
        push    DE              ; 'BDOS corrupts 99% of HOUSEHOLD
        push    BC              ; registers'.

        ex      de,hl           ; get current buffer position in DE
        call    set_dma_ptr

        call    write_line_to_file
        jr      nz,there_was_an_error
```

```
        POP     BC              ; restore our registers
        POP     DE
        POP     HL

        ADD     DE,HL           ; get next block in buffer

        DJNZ    write_buffer    ; until we have finished

;---------


FN_SET_DMA_ADDRESS      equ     1AH

set_dma_ptr:
        LD      C,FN_SET_DMA_ADDRESS
        CALL    BDOS
        RET

;----------------
```

Passing filenames via the keyboard

Up to the present point, filenames have been included in your program's source code. If you wanted to use a different filename after the program had been assembled, you would have had to start from scratch again by changing the filename and then re-assembling the program.

This situation would not be much help with programs that use different files at different times. For example, what would be the use of a text editor that accepts only one filename for a text file? Fortunately, CP/M allows you to pass filenames from the keyboard to a program when it is loaded by the Console Command Processor (CCP).

When you load a program called, say, SORT, you can pass the names of two files to it by typing a command such as

SORT B: INFILE.DAT OUTFILE.DAT

The CCP does two things with this information

- It loads the text 'B: INFILE.DAT OUTFILE.DAT' (the *command line tail*) into the default data buffer from address 0080H
- It then loads the file SORT.COM into memory

The area from address 0080H looks like

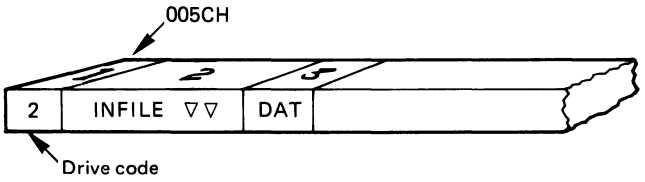


The first byte of the buffer contains the number of characters in the command line tail and the characters themselves follow it. Note that spaces are also transferred and are shown as '▽' in the diagram. Characters are translated into upper case ASCII, where necessary, and the unused part of the data buffer is left unchanged.

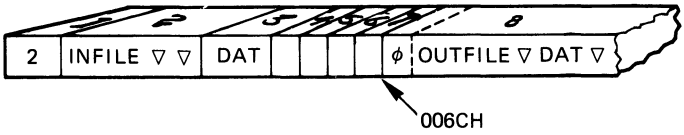
If you want to use the command line tail information you must move it from the default data buffer before you use the latter for file transfers.

If you use Set DMA Address to redefine the position of the data buffer, the command line tail will be left at address 0080H and it will not be overwritten.

Information from the command line tail is also stored in a default FCB. The first part is stored at address 005CH, in fields 2 and 3



Notice that the dots and colons have been removed. The second part of the command line tail (OUTFILE.DAT) is stored within field 8, at the address 006CH:



The second drive code (at address 006CH) takes the default value 0. Your filename OUTFILE is put immediately after this and the file type (DAT) follows. The rest of field 8 and field 9 are set to zero.

The CCP translates alphabetic characters into upper case so as to be consistent with CP/M's naming conventions. If the command line tail of SORT is blank, there will be no information stored from addresses 005CH and 006CH.

Note that you must move the second filename and file type (OUTFILE.DAT, in this case) to a separate FCB before you open the file that uses the first part of the FCB. If you do not, the Open File operation will overwrite this information!

Directory operations

There are a number of BDOS functions that affect complete files and their entries in the directory. These are

- Delete File (function 19)
- Rename File (function 23)
- Search for First Entry (function 17)
- Search for Next Entry (function 18)

Delete File (function 19) and *Rename File* (function 23) have the same effect as the equivalent CP/M console commands that delete and rename a file. Why would you want to use them inside a program? Well, suppose your program processes a file called FILE1 and always keeps a back-up of the file (called FILE1.BAK) as it existed before processing. What you might do is follow the procedure shown in figure 5.4.

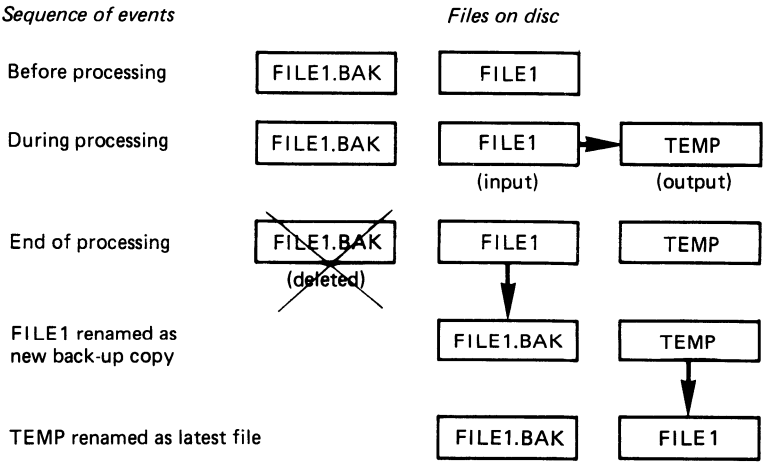


Figure 5.4 File back-up procedure

Here, you use FILE1 as input file and create another file (say TEMP) as output file. If your program run is successful, the program then deletes FILE1.BAK (using Delete File) and renames FILE1 as FILE1.BAK (using Rename File). It then renames TEMP as FILE1.

Delete File (function 19) is very similar to Create File. You load the filename into an FCB pointed to by registers DE, load 19 into register C and then call the BDOS.

Rename File (function 23) involves a little more work and the following piece of code shows it in use. You must set up the old filename in the first 16 bytes of the FCB and the new filename in the next 16 bytes.

```
TPA                     equ     100h
BDOS                    equ     5
default_drive           equ     0

null                    equ     0
clrscreen               equ     1FH
cr                      equ     0DH
lf                      equ     0AH
end_of_string           equ     '$'

        org     TPA
START:
        LD      SP,my_top_stack
        CALL    display_signon_msg

        LD      DE,rename_fcb
        CALL    rename_file
        INC     A                       ; check error return for -1
        JR      NZ,EXIT

there_was_an_error:
        CALL    display_error_msg
        JP      EXIT

;-----------------

RENAME_FCB:

RENAME_FCB.oldfile_name:
        DEFB    default_drive           ; currently selected drive
        DEFM    'TEMP    '              ; name of oldfile
        DEFM    '   '                   ; file type
        DEFB    0,0,0,0

RENAME_FCB.newname:
        DEFB    default_drive
        DEFM    'FILE1   '              ; new name for file
        DEFM    '   '                   ; new file type
        DEFB    0,0,0,0
        DEFB    0,0,0,0

;----------------
```

```
FN_RENAME_FILE  equ     17H

RENAME_FILE:
        LD      C,FN_RENAME_FILE
        CALL    BDOS
        RET
;-------------

FN_SYSTEM_RESET equ     0

EXIT:
        LD      C,FN_SYSTEM_RESET
        CALL    BDOS                    ; exit to CP/M

        RET                             ; just in case
```

Search for First Entry and Search for Next Entry are a little more sophisticated. They allow you to search the directory for the existence of one or more files and you might use them where your program needs to list the directory or part of it.

When you use *Search for First Entry* (function 17), CP/M tries to find a directory entry that matches the name you specify in an associated FCB. It will then return a value that tells you whether or not the entry has been found.

An extension that makes Search for First Entry much more powerful is its facility called *wildcards,* where you can use a *?* character in the FCB in place of any character. In this case, CP/M will take any character in that position as a match.

Suppose you want to list all of the files in the directory that are text files and have the file type .TXT. The following piece of code will do this:

```
TPA                     equ     100h
BDOS                    equ     5

length_fcb              equ     36.
length_filename         equ     8.
length_filetype         equ     3.
extent_position         equ     12.
default_data_buffer     equ     80H

default_fcb             equ     5CH
default_fcb.drive       equ     default_fcb
default_fcb.filename    equ     default_fcb+1
default_fcb.filetype    equ     default_fcb+9
default_fcb.extent      equ     default_fcb+12.
default_fcb.sys1        equ     default_fcb+13.
default_fcb.sys2        equ     default_fcb+14.
default_fcb.record_cnt  equ     default_fcb+15.
default_fcb.alloc_ptrs  equ     default_fcb+16.
```

```
default_fcb.current_rec equ     default_fcb+32.
default_fcb.random_rec  equ     default_fcb+33.

zero                    equ     0
clrscreen               equ     1FH
cr                      equ     0DH
lf                      equ     0AH
end_of_string           equ     '$'

        org     TPA
START:
        LD      SP,my_top_stack
        CALL    display_signon_msg

        CALL    CHECK_FOR_CMD_TAIL      ; have we started ok?
        JR      Z,no_cmd_tail_error

        CALL    ZERO_NON_FILENAME_PART_OF_FCB

        LD      DE,default_fcb
        CALL    SEARCH_FIRST_FILE_ENTRY
        INC     A                       ; test if found
        JR      Z,file_not_found        ;     -1 = not found

MORE:
        CALL    PRINT_FILENAME          ; print it.

        LD      DE,default_fcb
        CALL    SEARCH_NEXT_FILE_ENTRY  ; check if more entries
        INC     A                       ; test for -1 = not found
        JR      NZ,MORE                 ; and try again

ALL_DONE:
        JP      EXIT

;--------------

file_not_found:

        LD      DE,file_not_found_msg
        CALL    PRINT_STRING
        JP      EXIT

file_not_found_msg:
        defb    cr,lf
        defm    ' FILE NOT found'
        defb    cr,lf
        defb    end_of_string

;-----------------
```

```
ZERO_NON_FILENAME_PART_OF_FCB:

        LD      HL,default_fcb.extent          ; clear rest of FCB
        LD      B,length_fcb - extent_position
        LD      A,zero
1$:
        LD      (HL),A
        INC     HL
        DJNZ    1$

        RET

;-----------------

FN_SEARCH_FIRST equ     11H

SEARCH_FIRST_FILE_ENTRY:

        LD      C,FN_SEARCH_FIRST
        CALL    BDOS
        RET

;-------------

FN_SEARCH_NEXT  equ     12H

SEARCH_NEXT_FILE_ENTRY:

        LD      C,FN_SEARCH_NEXT
        CALL    BDOS
        RET
```

The BDOS function *Search for First Entry* finds the first entry that you want in the directory. If the command line contains ????????.TXT as the filename, the characters 'TXT' will be set up in the FCB as the file type and the filename will contain several '?' (wildcard) characters. CP/M will ignore the entire filename and just look for a file of the type .TXT. The subroutine PRINT_FILENAME then prints the filename on the screen.

If you are using wildcards and want to look for any more .TXT files, you can do so using the BDOS function *Search for Next Entry* (function 18). Its action is similar to that of Search for First Entry and an example of its use is also shown in the program above.

Using random access with files

Up to now you have written and read files sequentially. *Random access* is a way of getting at the sectors of a file in random order.

The rest of this chapter describes the more elementary aspects of using random access; to understand it fully you need to know how files are organised on disc. Thus, it is covered in more detail in chapter 8, after disc structures have been dealt with.

Random access of a sequentially written file

The easiest way of producing and using a random access file is to write it sequentially, just as you have done before, then read and update sectors in random order.

You have already written a file DATAFILE.DAT in sequential order. If you have, say, 10 sectors in it and you want to read sector 10, there would be no hardship involved in reading the file sequentially to get to record 10. However, if there are 1000 records in it and you want to read sector 900, reading 899 records to get to the one you want would be tedious. Even if the file is short, if you frequently want to access different sectors in random order, resetting the file and reading it sequentially would give a large overhead.

What you can do, instead, is open the file and use the BDOS function *Read Random* (function 33) to get at sectors in random order. This is similar to Read Sequential but takes an additional parameter – the sector number.

You set up the function in a similar way to the other disc operations: load the address of the file FCB into the register pair DE, load 33 into register C and then call the BDOS. However, you must also put the required sector number into a special part of the FCB (bytes 33 to 35) known as the *record pointer.*

The record pointer is shown in figure 5.5. When using random access you must always remember to allocate a full 36 bytes for the FCB to allow for this pointer. If you allocate only 33 bytes, you will find that the next 3 bytes of your program are overwritten.

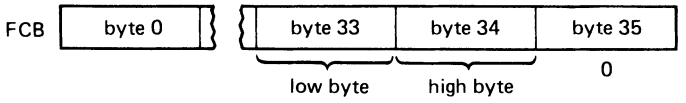


Figure 5.5 The record pointer

Bytes 33 and 34 of the FCB are a 16-bit unsigned integer and are set to the (two-byte) sector number. Byte 35 is set to zero. For example, if you want to read sector 900 (384H), you would set byte 33 to 84H, byte 34 to 3H and byte 35 to 0.

If your read operation is successful, the data is returned in the current disc data buffer and a zero value will be returned to your program in the A register. Any other value in the A register means that CP/M has been unable to read the sector

for one reason or another. These error returns will be described in more detail in chapter 8, after some other concepts have first been introduced.

The program listing below shows how Read Random is used. It opens a file DATAFILE.DAT which contains lines of text, prompts you for a line number and then displays the line. You have already produced this file; it contains a number of sectors, each of which contains up to one line of text.

```
TPA                     equ     100h
BDOS                    equ     5
default_data_buffer     equ     80h
default_drive           equ     0

null                    equ     0
clrscreen               equ     1FH
CTRL_C                  equ     3
cr                      equ     0DH
lf                      equ     0AH
end_of_string           equ     '$'

max_line_length         equ     126.

        org     TPA
START:
        LD      SP,my_top_stack
        CALL    display_signon_msg

        CALL    OPEN_DATA_FILE
        INC     A                       ; test error return for -1
        JR      Z,file_error

display_loop:
        CALL    PROMPT_FOR_LINE_NUMBER
        JR      C,LE_FINI

        LD      (datafile_fcb.random_rec),A  ; set record no to read
        CALL    READ_RANDOM_LINE_FROM_FILE   ; get line of text
        JR      NZ,file_error                ; test for error.

        CALL    WRITE_LINE              ; go and print it
        JR      display_loop            ; and round again

LE_FINI:
        CALL    CLOSE_DATA_FILE
        JP      EXIT

;-----------------
```

```
datafile_fcb:

        DEFB    default_drive           ; currently selected drive
        DEFM    'DATAFILE'              ; name of file
        DEFM    'DAT'                   ; file type
        DEFB    0,0,0,0,0,0,0,0         ; pad the rest with zeros
        DEFB    0,0,0,0,0,0,0,0
        DEFB    0,0,0,0,0
datafile_fcb.random_rec:
        defb    0,0,0
;---------------

FN_READ_RANDOM  equ     21H

READ_RANDOM_LINE_FROM_FILE:
        LD      DE,datafile_fcb
        LD      C,FN_READ_RANDOM
        CALL    BDOS
        OR      A                       ; set flags
        RET
```

Up to now we have just considered reading a previously written file in random order. We can now consider *replacing* an existing record using the BDOS function *Write Random* (function 34).

You set up the function in a similar way to Write Sequential: load the address of the file FCB into the register pair DE, load 34 into register C and then call the BDOS. Again, you must also put the required sector number into the record pointer part of the 36-byte FCB.

If you want to *update* an existing sector, you must first use Read Random to read the contents of the sector into the disc data buffer. You then change the data and write it back to disc using Write Random.

As with the sequential mode of operation, when you have finished with your file you should close it.

Writing sectors in random order

Suppose you want to build an employee record system for a business. In this system, each employee might have an employee number which corresponds to a sector on the file, and that sector holds the employee's record.

You might want to allocate a block of employee numbers to each department so that people can be identified more readily. In addition, each block of numbers might have 'holes' to allow for the addition of records later for new employees. For example

MANAGEMENT	has 5	staff and the numbers 0–9
PERSONNEL	have 6	staff and the numbers 10–19
SALES	have 2	staff and the numbers 20–25
ACCOUNTS	have 3	staff and the numbers 26–29
ADMIN	have 3	staff and the numbers 30–39
WORKSHOPS	have 40	staff and the numbers 40–200

The record layout is shown in figure 5.6.

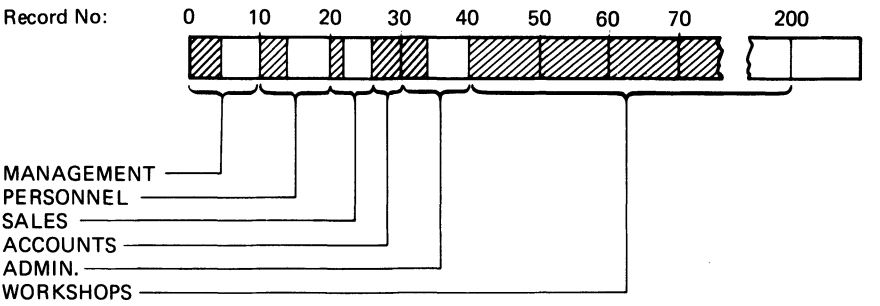


Figure 5.6 Employee record file

In figure 5.6, the shaded portions show records that have been written to the file. The unshaded portions are 'holes', which you can think of as records that have not yet been written. Now, how do we handle the 'holes'?

The easiest way would be to do it as we have previously: create a file sequentially and fill it with 200 'dummy' sectors containing zeros. We can then use Write Random to replace some of these sectors with the real employee data sectors.

However, a much easier way of achieving this structure is to use the BDOS function *Write Random with Zero Fill* (function 40). This is the same as Write Random, but it fills any preceding sectors that are not yet used with zeros. For example, suppose the first few records of your file are as shown in figure 5.7 (only the MANAGEMENT records have been written)

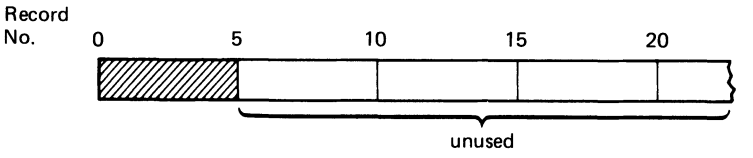


Figure 5.7 Adding records to the file

If you now want to add the first record for PERSONNEL, you can do this using Write Random With Zero Fill. This operation will write sector 10 but will also fill sectors 6 to 9 with zeros. The remaining 5 records for PERSONNEL can also be written using Write Random With Zero Fill, but the effect will be the same as for Write Random because there are no 'holes'.

Now, suppose we want to write the first record in SALES (record 20). Write Random with Zero Fill will write this record and fill the intervening records (16 to 19) with zeros.

Write Random with Zero Fill allows us to create a normal database file with few operations, and that file is the same as though it were written sequentially.

The previous discussion has been kept deliberately short so that you can understand the principles involved in using random access files. In practice, if you write files containing 'holes', you need to know more about how they are laid out on the disc, otherwise you will have problems. This aspect is covered in chapter 8, after the main disc structures have been described.

Error handling with files

CP/M may return to your program one of the error codes shown in table 5.1 as the result of a file operation. Error codes are returned in the A register.

Table 5.1 File handling error codes

Error code	*Meaning*
1	Reading unwritten data
2	Disc is full
3	Cannot close current extent
4	Accessing unwritten extent
5	Directory full
6	Byte 35 of FCB non-zero

Some of these codes will be more meaningful when you have read chapter 8.

Summary

- Each file has an entry in the disc's directory and this holds details such as the filename and position of the file on disc
- When you create a file you first use BDOS function *Create File* (function 22), write to it, then finally close it with BDOS function *Close File* (function 16)
- When CP/M creates or opens a file it uses information in a *File Control Block (FCB)* which is set up by you
- Data is read from disc and written to it as 128-byte blocks and these are put into a *disc data buffer*

- CP/M allocates a *default data buffer* at address 0080H
- You can change the address of the disc data buffer using BDOS function *Set DMA Address* (function 26)
- Files are of two types: text files and binary files
- Filenames can be passed to your program from the keyboard. Details are stored in the default data buffer
- Files can be read and written sequentially, or in random order. You can also append blocks to a sequentially written file
- CP/M may return to your program one of the error codes shown in table 5.1 as the result of a file operation. Error codes are returned in the A register

6 Disc Operations

Allocation maps	*Get Address of Allocation Map*	*Return Logged-in Drives*
Check vectors	*Get DPB Address*	*Scratchpad area*
Currently selected drive	*Get Read-only Indicators*	*Sector translate tables*
Directory buffer	*Logged-in drive*	*Select Disc*
Disc data structures	*Pre-allocated block map*	*Set File Attributes*
Disc Parameter Block (DPB)	*Reserved tracks*	*Write-protect Disc*
Disc Parameter Header (DPH)	*Reset Disc System*	
File protection	*Return Current Drive*	

The first section of this chapter introduces some of the terms used in disc handling and describes some of the basic disc handling functions. The second section describes how you can protect your discs and files from accidental overwriting. Finally, the third section tells you how to get hold of disc characteristics.

Introduction

Two terms are used frequently in connection with CP/M disc handling: the *currently selected drive* and *logged-in drives.*

The *currently selected drive* is used as the default drive for all file operations. It is the drive indicated by the prompt on your screen; for example

```
A>
```

When you type B:<RETURN> the currently selected drive changes to drive B.

You can determine the currently selected drive from your programs by using the BDOS function *Return Current Drive* (function 25). This returns the drive number in register A. You can also change the currently selected drive using the BDOS function *Select Disc* (function 14).

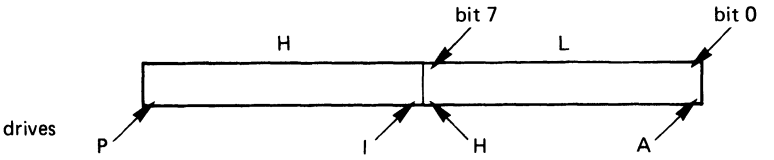
Logged-in drives are the drives that CP/M knows about. When you cold-start or warm-start your system, the only logged-in drive becomes drive A, the system drive. Whenever you access any other drive, that drive also becomes a logged-in drive. For example, if you type

```
PIP C:DATAFILE.DAT=RDR:
```

drive C also becomes a logged-in drive, even though the currently selected drive is drive A.

CP/M keeps in memory a set of tables that describe drives and their contents (they are covered in more detail later in this chapter). It knows about only one drive at any moment (the currently selected drive) but it can access the tables of all logged-in drives quite easily, if necessary.

You can find out which are the logged-in drives from your program by using the BDOS function *Return Logged-In Drives* (function 24). This returns a 16-bit value in the register pair HL, where each bit position corresponds to a drive using the scheme



If the appropriate bit is set, the drive is logged-in.

You can reset the currently selected drive to drive A and make this the only logged-in drive by performing a cold start. You can also do it from your program using the BDOS function *Reset Disc System* (function 13). This function also resets the default DMA address to 0080H.

If you perform a warm start (by pressing CTRL/C at the keyboard) this will make the drive that you were last using the currently selected drive and also log in this drive and drive A.

Protecting your discs and files

You can protect information on your discs by

- Using the physical write-protect mechanism of your system (covering or uncovering the write-protect notch as appropriate)
- Using the BDOS function Write Protect Disc (function 28) to write protect the entire disc
- Using the BDOS function Set File Attributes (function 30) to write-protect a specific file

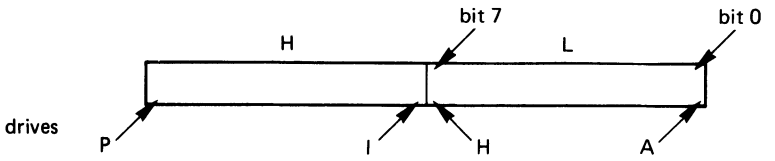
Physical write-protection over-rides all other protection mechanisms. If your discs are write-protected in this way, no program can write to them.

Write Protect Disc (function 28) allows you to protect the currently selected drive until one of the following occurs

- A cold start or a warm start
- Reset Drive (function 37) is used on this drive
- Reset Disc System (function 13) is called

Discs protected by Write Protect Disc are protected only against programs using CP/M BDOS functions. Programs that use BIOS or firmware routines can bypass this method of protection.

You can find out which discs have been protected using Write Protect Disc by calling the BDOS function *Get Read-Only Indicators* (function 29). This returns a value in registers HL where each bit indicates the status of a specific drive.



If a particular bit is set to 1, the appropriate drive is protected against writing by CP/M. Note that this function does not tell you the state of the write-protect notch on the disc.

To write-protect a file you use the BDOS function *Set File Attributes* (function 30). This writes to the most significant bit in byte 9 of the FCB (called the read-only file attribute). If the bit is set, the file is protected against writing; if it is clear, the file will not be protected. You should not change the read-only file attribute other than by using Set File Attributes.

You can tell if a file has been protected by using the BDOS function Search for First Entry (function 17) in your programs. This returns the directory entry of the file in the form of an FCB; you can then look at its status bit to find out whether or not the file is protected.

Getting hold of disc characteristics

The first part of this section describes the disc data structures in memory in general terms with a brief description of each one. The next 2 parts look at two of the most useful data structures for the applications programmer – the allocation maps and the disc parameter blocks. Finally, the sizing of allocation blocks is discussed.

Introduction to the disc data structures

This section describes how you can get at some of the information CP/M uses for its disc handling. The information includes disc sizes, details about the positioning of files on the disc and the number of directory entries on the disc. You would

not want to use it in all of your programs, but if you do need it (for example, to produce disc statistics), the information is not difficult to get at. You should be careful not to change it though, as you will confuse CP/M if you do not do this correctly.

Figure 6.1 shows the data structures that CP/M uses for its disc handling. These are all held in memory.

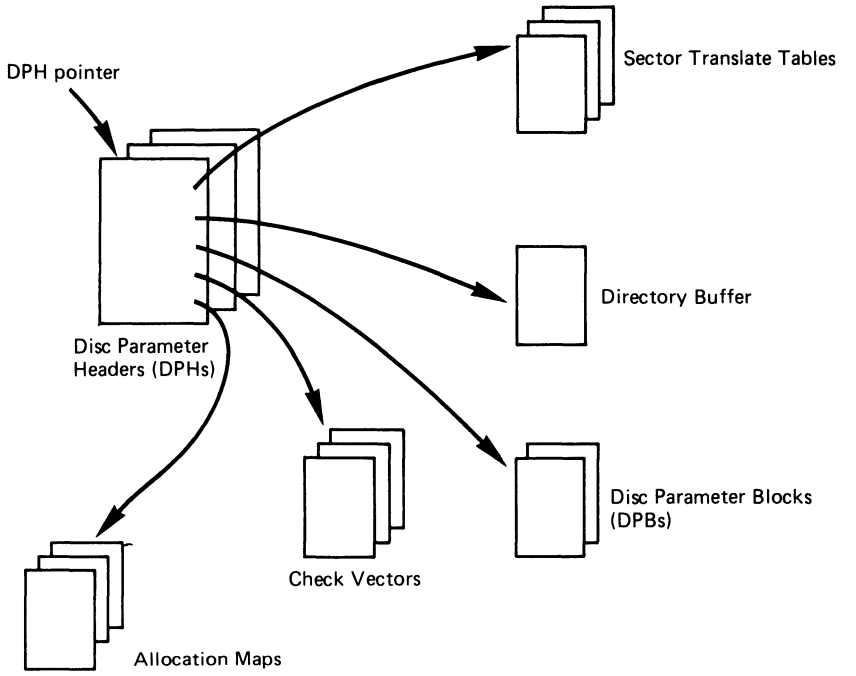


Figure 6.1 CP/M disc data structures

There is one *Disc Parameter Header (DPH)* for each logical drive, that is, one each for drives A, B, C, D, E, F, or however many drives your system has. However, the BDOS only knows about one drive at any moment: the currently selected drive. It maintains a pointer to the DPH of this drive so that it can get at the drive details quickly. When you perform a disc operation with an embedded drive reference to another drive, the BDOS

- Remembers the drive number of the currently selected drive
- Selects the drive referenced and changes the DPH pointer to point to the DPH of the new drive
- Performs the disc operation
- Reselects the original drive and restores the DPH pointer

Each DPH is predominantly a table containing the addresses of other data structures; these addresses are known as *pointers*. The pointers are shown in Figure 6.2.

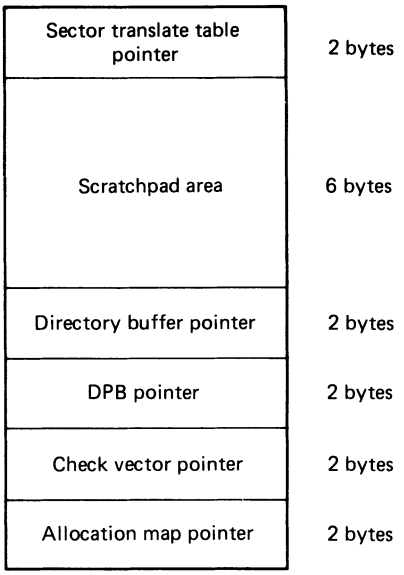


Figure 6.2 Disc parameter header

We will now look at the data structures that the pointers address.

There are *sector translate tables* for each type of disc (5.25 inch, 8 inch, etc.). They are used to translate logical sectors into physical sectors on the disc so as to improve the overall response of CP/M.

The *scratchpad area* is an area of memory reserved by the BDOS for its own use.

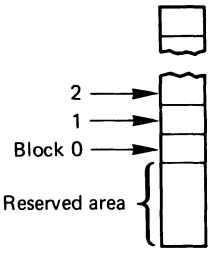
The *directory buffer* is an area used by the BDOS for directory operations. There is only one of these for the entire system.

The *check vectors* are areas of memory that are used only with exchangeable discs. They contain information used as a check against changed discs. CP/M keeps a type of 'checksum' in this area of all the directory entries of the disc. It can tell when a disc has been changed by scanning the directory and comparing the result with this 'checksum'. The length of the check vector depends on the number of directory entries. There is one check vector for each exchangeable drive and the pointer in the DPH directs you to this.

The allocation maps

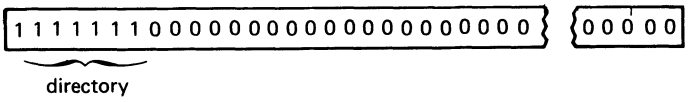
The smallest unit of file allocation on a disc is called an *allocation block* and this can be 1024, 2048, 4096, 8192 or 16384 bytes long, depending on the disc type and a number of factors described later in this section.

Each disc is divided into a number of allocation blocks with the first block starting immediately above the area on disc reserved for CP/M. This is called Block 0.



CP/M keeps a record of which allocation blocks on a disc are in use and which ones are not, and this is called an *allocation map.* It is basically a memory representation of what the disc looks like. There is an allocation map for every disc and it consists of a bit map, each bit of which corresponds to an allocation block.

When you log into a disc, CP/M builds up an allocation map for it. The directory consists of a number of pre-allocated blocks and these have their appropriate bits set (the first few allocation blocks in the map).



CP/M then scans the directory and sets bits in the allocation map for whichever allocation blocks are used by files.

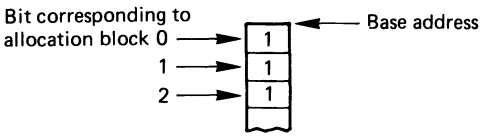
Suppose that we start off with an empty disc and the allocation map looks like the one just shown. Now suppose that your program tries to create a file on the disc. CP/M will search the allocation map for the first free allocation block and mark this as being used for your file. Further blocks are assigned to the file as required and they may or may not be contiguous, depending on whether or not other files are on the disc. Details of the allocation blocks used by a file are kept in the file's directory entry, and when you use Open File (function 15) they are copied into bytes 16 to 31 of the FCB. For discs with allocation blocks of 1024 bytes, this FCB area consists of 16 one-byte integers. For larger allocation blocks, it consists of eight 16-bit integers.

Thus, assuming that we are using 1024 byte allocation blocks, if a file uses allocation blocks 20, 21, 25 and 26, these FCB bytes will contain

Byte of FCB	*Contents*
16	20
17	21
18	25
19	26
20	0
21	0

Your programs can read this information from the FCB at any time when the file is open, *but they must not change bytes 16 to 31.*

If you want to look at the entire allocation map for a disc, you can do so by using the BDOS function *Get Address of Allocation Map* (function 27). This returns the base address of the current disc's allocation map.



To be safe, you should first use Reset Disc System, then Select Disc, and finally Get Address of Allocation Map. This is the only way you can ensure that the allocation map is accurate.

One of the example routines in appendix A uses Get Address of Allocation Map to calculate the amount of free space on the current disc. This is similar to the STAT utility provided with CP/M, but unlike STAT you can use it from your own programs.

You may also need to know the following information about the disc structures.

- The size of each allocation block
- The number of allocation blocks on the disc
- The number of allocation blocks occupied by the directory

To get this information you have to access the current disc's DPB and we shall look at this area in the next section.

The Disc Parameter Blocks (DPBs)

Disc Parameter Blocks (DPBs) are tables of memory that describe the characteristics of discs on your system, and there is one DPB for each disc type (for example,

5.25 inch, 8 inch, single sided, double density, etc.). Hence, if you have several discs of the same type on a system, several DPHs could point to the same DPB.

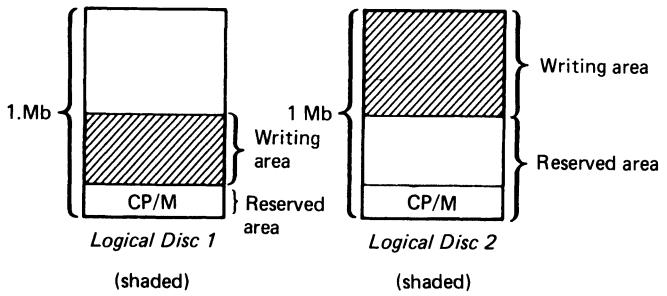
No. of 128 byte logical sectors per track	2 bytes
Field A	1 byte
Field B (see text below)	1 byte
Field C	1 byte
Disc size in allocation blocks - 1	2 bytes
No. of directory entries - 1	2 bytes
Pre-allocated block map	2 bytes
Size of check vector	2 bytes
No. of reserved tracks	2 bytes

Figure 6.3 Contents of a DPB

The contents of a DPB are shown in figure 6.3. You can get at the start address of the DPB for the currently selected drive using the BDOS function *Get DPB Address.*

Some of the contents are easily understood: the number of sectors on each track of the disc, the maximum number of directory entries that can be stored on the drive and the size of the check vector in bytes.

The *number of reserved tracks* normally refers to the number of tracks used to store the operating system on the disc. However, it may also be used to split a large disc (for example, a Winchester) into several logical drives. For example, if you wanted to divide a 1 Megabyte disc into two logical 500 Kilobyte drives, the arrangement is



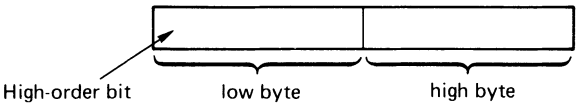
You can determine the allocation block size in bytes from the field B shown in figure 6.3. The relationship is

Allocation block size = (B + 1) * 128

and you can determine the size of the disc in bytes using this figure and the fifth field of the DPB: *disc size in allocation blocks – 1.* The relationship is

Disc size = (Allocation block size) * (disc size in allocation blocks)

The *pre-allocated block map* is a 16-bit field that describes how many allocation blocks are pre-allocated for directory entries.



Bits of this field are set from the high-order bit of the low byte. Thus, if four directory blocks are filled, the four high-order bits of the low byte are set, giving it the value FOH, and the high byte is set to 0.

You might also want to calculate the size of the allocation map in bytes. You can get this from the DPB as follows

Allocation map size = disc size in allocation blocks/8

Allocation block sizes

An allocation block is the smallest unit of file allocation and its size depends on a number of factors.

If the allocation block size were 128 bytes, the size of each sector written by your program, then CP/M would have to remember a lot of information about each file. The disc direction would be large and so would the allocation map and check vector in memory. In practice, the smallest possible allocation block is 1K.

On the other hand, if the allocation block size is large (say 16Kb) and you write a file containing only 100 bytes of data, then over 15Kb of disc space would be wasted (it could not be used by other files). If you have 100 such files of this type, the amount of disc space that these occupy for different sizes of allocation block is as follows

Allocation block size	*Space occupied by files*
16K	1.6Mb
8K	800Kb
4K	400Kb
2K	200Kb
1K	100Kb

There is obviously a lot more wastage for large allocation blocks!

The size of an allocation block is a compromise between these two main factors: disc space wasted and memory used. For efficient use of disc space, 1K is the ideal allocation block size. For efficient use of memory, 16K is the ideal. We will now look at some of the more detailed factors involved in deciding on the size of an allocation block.

The allocation pointers used in the FCB can be of two types

- 16 eight-bit pointers, each taking a value from 0 to 255
- 8 sixteen-bit pointers, each taking a value from 0 to 65535

If we use 1K allocation blocks and 8-bit pointers, this limits the maximum disc size to 255K. If we use 2K allocation blocks and 16-bit pointers, the discs supported can be much bigger. In practice, 8-bit pointers are used only with 1K allocation blocks and for discs up to 255K in size. Blocks of larger size use 16-bit pointers.

One other factor to consider is the way the BIOS handles sectors of data. Some BIOSs optimise disc operations to read a whole allocation block into memory in one step. Sectors are then blocked and deblocked from this.

Summary

- The *currently selected drive* is used as the default drive for all file operations. It is the drive indicated by the prompt on your screen
- CP/M remembers the details of all drives that it accesses and these are called *logged-in drives*
- You can find out which is the currently selected drive from your program with *Return Current Drive* (function 25)
- You can change the currently selected drive using *Select Disc* (function 14)
- You can find out which discs are logged in by using *Return Logged-in Drives* (function 24)
- You can reset the currently selected drive to A and make this the only logged-in drive by using *Reset Disc System* (function 13)

- *Write Protect Disc* (function 28) protects the currently selected drive against programs using BDOS functions
- *Get Read-Only Indicators* (function 29) tells you the status of a specific drive
- *Set File Attributes* (function 30) lets you write-protect a file
- There is one *Disc Parameter Header (DPH)* for every logical drive but the BDOS only knows about one at any time: the one for the currently selected drive. The DPH is predominantly a table with addresses of other data structures
- The smallest unit of file allocation on a disc is the *allocation block.* Each allocation block has a bit position in an *allocation map* which tells CP/M where blocks are free
- You can access the allocation map base address using *Get Address of Allocation Map* (function 27)
- *Disc Parameter Blocks (DPBs)* describe the characteristics of discs on the system
- The size of an allocation block is a compromise between disc space wasted and memory used. For efficient use of disc space, 1K is the ideal allocation block size. For efficient use of memory, and a realistic size of directory, 16K is the ideal size

7 Debugging your Programs

If something can go wrong, it will!

Murphy's Law

We have designed this chapter so that the first section is independent of the computer language that you use. It covers the general principles of debugging and describes a philosophy that applies as much to the home programmer using BASIC as it does to the masochist using machine code.

The second section is meant for those assembler programmers who intend to use the debugging tool ZSID. It explains how to use ZSID for the most common actions needed when you are debugging a program. You may, however, find it useful even if you are using the 8080 debugging tool DDT, and we have included appropriate comments on DDT where necessary.

General principles

How do bugs get into programs?

There are three types of bug that can get into your programs:

- Those that are introduced as a result of design errors. For example, you may be using the wrong algorithm
- Those that are introduced by mistyping. This is most likely to occur when you first type in your program source, because of the sheer quantity of code involved. You might, for example, type GOTO instead of GOSUB in parts of a BASIC program
- Those caused by an unexpected event, such as the corruption of a program on disc

The chances of you getting the last type of bug in your program are slim, but if you cannot find any logical reason for your program failing, do not be afraid to look outside it and analyse your own behaviour as well as that of your equipment. If the problem occurs a number of times, look for a pattern.

Keeping bugs out of your programs

The old adage 'Prevention is better than cure' could almost have been written with debugging in mind. Careful thought in the design and coding processes will not eliminate all of the bugs, but it will remove most of them.

The most significant step in removing bugs is to design your program correctly in the first place. Do not forget that this also involves the design of test facilities and test data.

Assume that everything will go wrong. Design contingencies into the program; test for the expected, the unexpected and the impossible. If you do not take action for a particular error, you should at least report it.

Design errors can be trapped by talking through your design with other people and then getting them to check it over on paper themselves. The first method clarifies your own understanding of how the program works and helps *you* to spot errors. The second gives you a check on your design by someone with a different mind (they probably will not make the same assumptions as you).

Most people cannot concentrate for more than 20 to 40 minutes at a stretch and they will benefit from a 5 minute break after this time. Bear this in mind when you are slogging at your design, coding, testing, whatever.

In particular, when you type your program source into the computer, do it in sensible 'chunks' with frequent breaks for coffee and a chat. You are then less likely to make mistyping errors. When you have typed it all in, run it through the assembler/compiler and let that spot the obvious mistakes for you. Then check it by eye, comparing it with the hand-written version. This may be boring but it will save you a lot of wasted time over the years; you may even see some obvious mistakes in your coding.

Trapping the ones that got away!

Bugs that make the program crash are the most obvious ones. The ones that do not crash it but merely change its effect are the worst type: they are the hardest to track down because their effect may not be noticed until much later in the program. To trap this type you need to use a lot of logical thought and be methodical.

Use lots of diagnostic statements. Add diagnostic message printouts to each module to tell you that it has been entered and print out any important variables that it uses. You will be surprised at the number of modules, subroutines, procedures and functions that do not actually perform as you thought they did, or are not invoked when you thought they would be.

Try to isolate any problems into as small an area of code as possible and single-step through critical areas. Reduce the problem to a particular module, then look carefully at it. Before diving into it make sure that it has its correct inputs. Do not forget that this module may rely on data provided by another module at a considerable distance away.

Is data modified where you expect it to be? Is it modified anywhere else? Trace the module through and check at each stage that the data is what you would expect.

Look carefully at

- Conditionals (IF statements, etc.)
- The use of pointers
- Register dependencies
- The effect of interrupts

Look at the state of the stack. You may find it useful to fill the stack area of memory with a particular pattern before it is used so that you can check out unexpected use of the stack. Check that the stack pointer is never above the top of the stack (a POP without the corresponding PUSH). Check also that the stack does not overflow into the area below the stack bottom. This can be caused by a PUSH without the corresponding POP, a CALL without a corresponding RET, or overuse of the stack.

When you find a bug it is better to fix it before checking for other bugs, otherwise, you may later be checking another effect caused by the same bug! However, strictly speaking this may mean re-running all tests from scratch again. In practice, you need to use a little common-sense: for example, run only those tests for the part of the program that has been changed by fixing the bug.

When you correct a bug, make sure you do not introduce a new variable with the same name as one already in use. If it is not easy to fix the bug, do not be afraid to scrap a piece of code and start again.

Update your program listing with a pen as you make each change; it is easy to get out of step and think you have introduced a change, or forget that you have done something.

Give your program a version number, and when you re-assemble the program, change the version number. Also, try to keep the comments up to date. There is nothing worse than having half a dozen copies of a program which claim to be the same thing but are all slightly different. If you are interrupted and have to come back to your debugging later, this will give you a big headache!

However, maintain old versions of the program so that if something goes wrong, or an old fault recurs, you can always wind back to a previous state and start again.

Getting rid of the rest by testing

It is better if someone other than the designer tests a program: the designer is more likely to concentrate on the areas that proved difficult in the design process and skimp on the rest.

You should design your test data so as to take the program through every branch; you should also predict the results. In particular, check boundary conditions and the parts that are rarely carried out: these usually occur at

conditionals (IF statements, or the equivalent in the language you are using). Compare the test results with the predicted results. If you cannot easily predict results, can you run your program in parallel with an existing system and use its inputs/outputs?

Structure your testing according to the circumstances. You need to test individual modules as well as test the whole program. There are two approaches

- Bottom-up testing: this starts by testing the primitive modules, writing a test frame to exercise them to their limits one by one, then building up to the entire program
- Top-down testing: this checks the logic of the main program first, with dummy modules to do the actual work

Both these methods have their merits and work best in different circumstances. Top-down testing allows you to check the main logic and have something at an early stage which behaves similar to the finished product. Bottom-up testing may be more helpful when debugging I/O drivers or where you are unsure of a particular technique. There is often more work involved in writing bottom-up test drivers than in writing top-down dummy routines.

Debugging under CP/M

There are three main debugging facilities available under CP/M: SID, DDT and ZSID.

All of them have features in common, but ZSID offers the best facilities for the Z80 programmer. It assembles and disassembles code using Zilog mnemonics, whereas SID and DDT handle 8080 code with Intel mnemonics. ZSID can also handle files on any drive, whereas if you are inspecting files using DDT, the files must be on the default drive.

For the purpose of this chapter, we will concentrate on ZSID. It is not our intention to describe its facilities in detail. Rather, we will look at the main functions that you would want to perform using a debugging tool and explain how ZSID handles them. They are as follows

- Loading a program (or ZSID) from disc
- Setting breakpoints and running the code
- Displaying the register set
- Displaying an area of memory
- 'Patching' data
- 'Patching' a program
- Disassembling a piece of code

The following sections describe each of these functions in turn. In each section, all items typed in by you are set in bold.

Loading a program (or ZSID) from disc

If you merely want to load ZSID, you can do this by typing

ZSID

You can load a program from disc via ZSID by typing

ZSID ***filename***

where filename is of the type PROG.COM. Thus, you could load the sample program of chapter 2 by typing

ZSID PROG.COM

In both cases, you will get the prompt

#

and you can now type ZSID commands.

Setting breakpoints and running the code

You can run a piece of code using the G command. For example, the following command runs the code from address 100H

G0100

You can also put into the code a single *breakpoint* (a place where the program will halt) using the G command. For example, the following will insert a breakpoint at address 110H and run the program up to that point

G0100 0110

Once the program has stopped, you can inspect the registers and data areas. If you want, you can run it from the breakpoint by typing

G

If you want to put more than one breakpoint into the program, the G command is not very efficient. Instead, you can use the S command, which 'patches' memory. You need to put an RST 38H instruction (FFH) into each breakpoint address and you can do this at addresses 0110H, 0120H and 0130H as follows

S0110
0110 CO **FF**
0111 DO .

S0120
0120 CC **FF**
0121 11 .

S0130
0130 21 **FF**
0131 11 .

You can run the program between these points using the G0100 command as described above.

Displaying the register set

You can print the current contents of the registers by typing

X

The display will look something like

```
----- A=00 B=0009 D=0200 H=0000 S=1000 P=0108
----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 CALL 0005
```

Most of this information is fairly obvious; however, some is not and needs a little further explanation. On the top line, the first item shows the value of the A register. The second item (B=0009) shows the contents of the B and C registers (B is 00 and C is 09). Similarly, the third and fourth items show the contents of the DE and HL register pairs. The items on the second line show the values of the alternate register set.

Displaying an area of memory

Occasionally, you will want to display a table or data area, and you can do this using the D command. For example, the following will display a screenful of memory starting at address 0080H, the default data buffer

D0080

```
0080:  C3 03 D9 41 01 C3 00 A9 AF 3D C9 40 8F 50 00 E0
        .  .  .  A  .  .  .  .  .  =  .  @  .  P  .  .
0090:  50 50 50 50 41 41 41 41 42 42 42 42 43 43 43 43
        P  P  P  P  A  A  A  A  B  B  B  B  C  C  C  C
00A0:  C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3
        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
00B0:  C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3
        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
00C0:  8F F0 8F 70 8F 71 8F 70 8F 70 8F 70 00 20 20 20
        .  .  .  p  .  q  .  p  .  p  .  p  .
```

You can display the next screen merely by typing

D

'Patching' data

The S command allows you to patch a block of data into memory. We have used it before to insert one byte of data (FFH) which forms a breakpoint. If, instead of typing the full stop after FFH, we continued, we could have typed data into subsequent locations. For example, the following instructions will zeroise the first five bytes of the default data buffer

```
S0080
0080 01 00
0081 01 00
0082 11 00
0083 21 00
0084 66 00
0085 16 .
```

With SID and ZSID only, you can type in ASCII characters using the S command if you precede them with a " character. For example

```
S0080
0080 00 00
0081 00 "THIS IS TEXT
008D 60 .
```

Note that the next prompted address is set to beyond the end of the character string.

'Patching' a program

The A command of ZSID allows you to 'patch' a piece of code into memory using Zilog mnemonics. It is really only useful when patching a program. Entering an entire program is best done via an assembler.

You can assemble a piece of code from address 100H by typing the following, then a carriage return

```
A0100
```

ZSID will then print out the address 0100 at the beginning of the line and you can type in a line of code, followed by a carriage return. ZSID will then print the next address, and so on until you finish by just typing a full stop followed by a carriage return.

The following instructions put the program example used in chapter 2 into memory. Try them out for yourself.

```
A0100
0100 LD  SP,1000
0103 LD  DE,200
0106 LD  C,9
0108 CALL  5
010B JP  0
010E .
```

To run the code, you need to put a string of text into memory at 200H. You can do this as follows

```
S0200
0200 C3 "Hello world!$
020E 1E .
```

Now, you can run the program by typing

```
G0100
```

Disassembling a piece of code

You can disassemble a block of code in memory (print it in mnemonic form) using the L command of ZSID. For example, typing the following would disassemble the code you typed in previously with the A command

```
L0100
0100 LD  SP,1000
0103 LD  DE,200
0106 LD  C,9
0108 CALL  5
010B JP  0
010E . . .
. . .
. . .
```

The L command has several forms. The one used above is the most common and it will print a screenful of disassembled code. If you want to print the next screenful you merely need to type

```
L
```

Summary

- There are three types of bug that can get into your programs

  Those caused by design errors
  Those caused by typing errors
  Those caused by an unexpected event such as corruption of your program on disc

- Careful thought during design and coding can eliminate most bugs
- It is better if someone other than the designer tests a program
- Structure your testing

8 Random Access Files

Allocation block	*Current extent number*	*Extent*
Allocation pointers	*Directory entry*	*Opening a new extent*
Creating a new extent	*Error codes*	*Sector count*

Chapter 5 explained the basic principles of using random access files. This chapter explains more of the theory so that you can avoid the pitfalls that exist with random access under CP/M.

The first section describes how files are written to disc using sequential access. This is an easy way to start and is a useful stepping stone to the next section, where we explain how random access files are written.

Creating sequential files

As we discovered in chapter 6, a disc is split into 2 parts

- The area reserved for CP/M
- A number of equal-sized allocation blocks used for the directory and your files

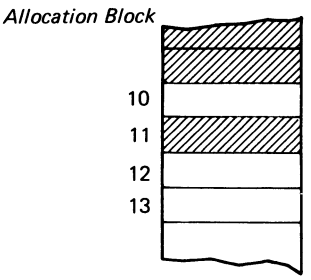
In the simplest case, an allocation block is 1024 bytes in size and is divided into eight 128-byte sectors. For the present, we will consider allocation blocks of this size.

When your program writes a file, CP/M allocates disc space to it in terms of allocation blocks. Your program may write only 10 bytes of data before closing the file, but CP/M will still give it 1 allocation block. When 16 allocation blocks are allocated to a file, they are grouped together and called an *extent.* Each extent is controlled by a directory entry.

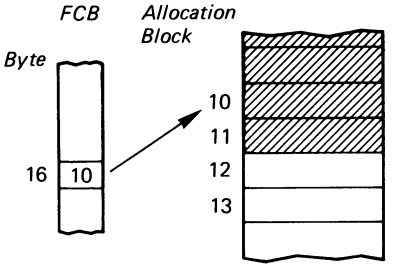
All file operations are controlled by the FCB and we are interested in 3 areas in particular

- The *allocation pointers* (bytes 16 to 31)
- The *sector count* within an extent (byte 15)
- The *current extent number* (byte 12)

Suppose you want to write a file sequentially and the disc has the following layout before you start (shaded areas show allocation blocks that are already in use by other programs)

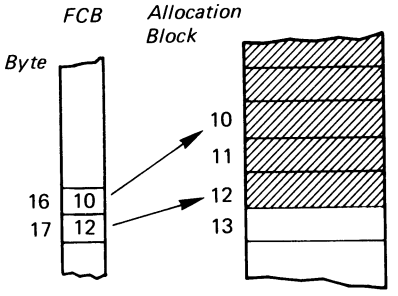


If you now create a file and write the first sector, CP/M will look for the first free allocation block and put the allocation block number (10) in the first FCB pointer as shown below. The current extent number (byte 12 of the FCB) and the sector count (byte 15 of the FCB) are both initially set to zero.



When your program writes sectors of data to the file, sector 1 goes into the first 128 bytes of allocation block 10 and the sector count of the FCB is set to 1. Sector 2 goes into the next 128 bytes of the allocation block and the sector count is set to 2. This continues up to and including sector 8, at which point an entire allocation block will have been filled.

When your program now writes sector 9 to disc, CP/M looks for the next free allocation block on the disc, finds that it is allocation block 12, and puts 12 into the second allocation pointer of the FCB as follows



The sector count is then incremented to 9.

Now, suppose that the file becomes so large that all FCB pointers are used up and the last allocation block given to the file has been filled. When the next sector is written to disc, CP/M writes the allocation pointers to the file's directory entry and creates a new extent for the file. Basically, this involves

- Creating a second directory entry for the file
- Resetting the sector count to zero
- Updating the current extent number (byte 12 of the FCB) to 1
- Changing all allocation block pointers in the FCB to zero

When CP/M selects the next free allocation block for the file, it puts the allocation block number into the first pointer of the FCB and the process starts again.

This process is called *creating a new extent.* When you read a large file, a similar process occurs and this is called *opening a new extent.*

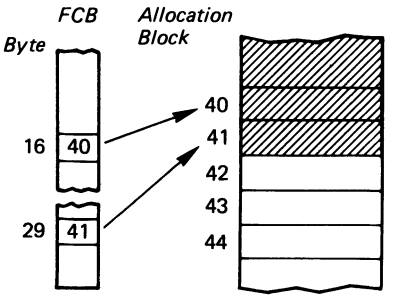
If your disc system uses 2048 byte allocation blocks the process is similar, but remember that the FCB now contains eight 16-bit pointers. Since each allocation block now contains 16 sectors, a new allocation block is opened after 16 sectors have been written. Also, a new extent is opened after 8 allocation blocks have been written.

Systems with allocation blocks of 4096 bytes or more have a similar but more complex mode of operation and are not described here.

How random access files are allocated

Suppose you want to write the records of a file in random order and they are each 1 sector in size. Again, we will assume an allocation block size of 1024 bytes.

If you want to write the first record, the first allocation pointer in the FCB will be given a value. Suppose it points to allocation block 40; then the data of record 1 will occupy the first 128 bytes of allocation block 40. If you now write record 5, the data will go into the fifth 128 byte part of allocation block 40.



If you want to write record 240, assuming that the file were contiguous and your system uses 1024 byte allocation blocks, this would map onto the 30th allocation block of the file. We arrive at this result using the following relationship

Allocation block no = record number/(size of allocation block in sectors)

The allocation block number is the whole number (integer) part of this expression; the remainder is the sector number in the allocation block. Thus, in the case of record 240, we get the result 240/8, or 30.

What happens is this

- The first directory entry is written to disc with allocation pointers 2 to 15 still set to zero
- The current extent number is updated to 1
- A new set of allocation pointers is created in the FCB and they are preset to zero

CP/M then searches for the next free allocation block (41) and gives this to your file, setting the 14th allocation pointer of the FCB to 41 (the next free allocation block).

Record 240 is written to the first sector of allocation block 41 and the sector count is updated.

Now think about this: the file is at least 240 records in length, but so far it occupies only 2 allocation blocks. If you try to read (say) record 130 or record 500, CP/M would in each case return in register A the error code 1, which means 'Reading unwritten data' (see table 8.1). Record 130 would occupy a place in the first allocation block of extent 1, if it had been written. Record 500 is beyond the end of the file and thus would have no allocation block as yet.

CP/M will attempt to warn you if you try to read unwritten data in the following circumstances.

- When the sector you specify is beyond the end of the file
- When it is in an unwritten allocation block
- When it is in an unwritten extent

If the sector is in an unwritten allocation block, you need to be especially careful. Suppose, for example, that you write only records 1 and 12 to a file. If you now try to read record 13, CP/M will return an error code. However, if you try to read any one of records 2 to 11, no error code will be returned, even though you have not written these records. Instead, CP/M will return whatever garbage is on the disc in the appropriate sector positions.

You must be careful when writing the records of a file in random order: if you have not written *all* of the records in the file, you will encounter problems when you read records from it. It is much safer to write the file sequentially and fill it with empty records, or use Write Random with Zero Fill.

Table 8.1 File handling error codes

Error code	*Meaning*
1	Reading unwritten data
2	Disc is full
3	Cannot close current extent
4	Accessing unwritten extent
5	Directory full
6	Byte 35 of FCB non-zero

Summary

- A disc is split into 2 parts: the area reserved for CP/M and a number of equal-sized allocation blocks used for the directory and your files
- Allocation blocks can be 1K, 2K, 4K, 8K or 16K in size
- When a file is written, CP/M allocates space to it in terms of allocation blocks. Even if the file contains only 10 bytes of data, it will still be one allocation block in size
- File operations are controlled by the FCB and 3 areas are of interest

  The allocation pointers (bytes 16 to 31)
  The sector count (byte 15)
  The current extent number (byte 12)

- Before a new extent is created or opened, the allocation pointers in the FCB are first written to a directory entry. An additional directory entry is created for each new extent
- The sector count is incremented when each sector is written, and resets to zero when the first sector of a new extent is written
- The current extent number is incremented when a new extent is opened
- CP/M will attempt to warn you if you try to read unwritten data in the following circumstances

  When the sector you specify is beyond the end of the file
  When it is in an unwritten allocation block
  When it is in an unwritten extent

9 *Using BDOS Functions from High Level Languages*

Accessing machine code	*PEEK*
BASIC	*POKE*
Getting data back from a machine code routine	*Stack*
Pascal	*Where to put machine code in memory*
Passing parameters to a machine code routine	

Why should you want to access BDOS functions from a high level language? Well, the majority of high level languages are designed without a specific operating system in mind, and some of the operations you might want to carry out will only be possible via the operating system itself.

For example, you might want to scan the directory for the first occurrence of a filename, list the directory itself, or find out the size of a file. Most high level languages will not allow you to do these things directly, but you can do them via BDOS functions.

High level languages may not specifically give you access to BDOS functions or other facilities of CP/M. However, most of them allow you to access machine code and you can do most of the things you want via this medium.

The interface between the high level language and machine code varies depending upon the language and the specific implementation. However, it is usually well-defined. In all cases, you need to know four things

- Where to put the machine code in memory
- How to access the machine code
- How to pass parameters to it
- How to get information back from it

Your high level language program will need to

- Set up any parameters needed
- Call the machine code routine
- Pick up any answer returned

Looking at the problem from the machine code end, your routine needs to do the following

- Preserve the environment (save registers and the return address)
- Pick up any parameters passed
- Perform the operation
- Return the results
- Restore the environment (reload registers and the return address)

In the rest of this chapter we will look at the interface with machine code of two common languages: BASIC and Pascal.

Accessing machine code from BASIC

BASICs in general do not have good facilities for accessing the operating system or getting at machine code. They usually allow you to

- Put a byte into a memory location (POKE)
- Look at a byte in a memory location (PEEK)
- Call a machine code subroutine or function

The way in which you do the last of these operations depends on the type of BASIC. For example, Microsoft BASIC allows you to call a routine at the address A000H, say, in the following way

```
CALL &A000H
```

If you want to put a machine code routine at this address, you can reserve space for it when you load a Microsoft BASIC program by typing something like the following

```
BASIC MYPROG/M:A000
```

This loads a BASIC program from the file MYPROG and reserves an area from address A000H to the operating system for your use. You can then get your BASIC program to POKE the machine code into this area and CALL it.

With the type of BASIC you use, it may be that you can call only one machine code routine, so you must use this routine as the starting point for a number of operations, passing information to it from BASIC to tell it which operation you want performed.

If you want to pass information to a machine code subroutine, you may have to POKE specific memory locations. Similarly, you might have to PEEK at memory to get information back from it. Alternatively, you may be able to do both via a user function.

Suppose you want to use the BDOS function Search for First Entry (function 17). You may have to POKE the contents of the FCB into memory, then call a user function to get at the machine code routine. The machine code routine

knows where the FCB is and calls the BDOS to do a Search for First Entry, passing the result back to BASIC.

Accessing machine code from Pascal

The method of accessing machine code from Pascal is similar to that of other 'professional' computer languages, for example, FORTRAN, C and BCPL. You define one or more subroutines, procedures or functions that are EXTERNAL to the main program and then link these to the main program via a linker.

One way of generating the final program is to compile the Pascal source, assemble the machine code as a separate operation and then link the two together.

Pascal passes parameters in a well-defined way. For example, with ProPascal, the parameters to a procedure are passed on the stack. They are pushed on to the stack with the first parameter first, the next parameter second and so on. If, for example, we use the following procedure call

```
PROCEDURE  fred (par1 par2 par3);
```

and 'fred' is EXTERNAL, par1 is first pushed onto the stack, then par2 is pushed on top of it, then par3. Next, a call is made to the machine code routine and, as a result, the return address is left on top of the stack.

In the machine code program, you must POP the return address from the stack and save it somewhere. You can then POP the parameters in turn, starting from par3.

The size of the parameters is defined for the particular language; for example, an integer may be 4 bytes long, so you have to POP 2 words off the stack. You should check the details of the appropriate language to find out how different values are passed.

As well as passing your machine code program the value of a variable, Pascal may pass it the address of the variable. Thus, it can act upon the actual variable and, when you return to Pascal, the value will have been updated.

The program that follows shows how to access a machine code function (fsize) from ProPascal. The Pascal program passes an FCB to fsize which then uses the BDOS function Compute File Size to work out the size of the named file. This value is passed back to the program and, finally, printed. A listing of the function FSIZE can be found in appendix A.

```
PROGRAM size;
(* This program computes the size of a file *)

CONST
  size_of_a_record      = 128;

TYPE
  byte          = 0..255;
```

```
  fcb           = record
                    drive_code  : byte;
                    fname       : array[1..8] of char;
                    ftype       : array[1..3] of char;
                    buffer      : array[0..26] of byte;
                  end;
VAR
  index                 : integer;
  file_ctrl_block       : fcb;
  filespec              : string;
  file_size             : INTEGER;

FUNCTION fsize(an_fcb : fcb) : INTEGER;
EXTERNAL;

(* This function is written in m/c code and returns either the size
of the specified file in records if it exists or -1. It uses CP/M BDOS
function FN_COMPUTE_FILE_SIZE. *)

BEGIN
  WRITELN('Compute file size program      v1.0b');
  WRITELN;
  WRITE('Please enter name of file ?');
  READLN(filespec);

  with file_ctrl_block do
    begin
      drive_code := 0;
      for index := 1 to 8 do
        fname[index] := filespec[index];
      for index := 1 to 3 do
        ftype[index] := filespec[index+8];
    end;

  file_size := fsize(file_ctrl_block) *  size_of_a_record;
  IF file_size < 0 THEN WRITELN('*** File does not exist ***')
    ELSE
      BEGIN
        file_size := file_size;
        WRITELN;
        WRITE(filespec);
        WRITELN(' is ',file_size,' bytes long');
      END;
END.
```

Summary

- A specific high level language may not let you access operating system functions but it will usually let you write machine code routines. You can then access the operating system functions from these
- When you write such a routine you need to know the following

  Where to put the machine code in memory
  How to access the machine code
  How to pass parameters to it
  How to get information back from it

- Your high level language program will need to

  Set up any parameters needed
  Call the machine code routine
  Pick up any answer returned

- Your machine code routine will need to

  Preserve the environment (save registers and the return address)
  Pick up any parameters passed
  Perform the operation
  Return the results
  Restore the environment (reload registers and the return address)

Appendix A: Example Programs

This appendix contains a number of programs that illustrate the use of BDOS functions. Most of them are self explanatory but the following notes describe some of their features.

Reaction timer program

This program uses routines described in chapter 4. It waits for you to press the space bar and then checks how long it takes you to press another key.

Get Console Status is used to break out of the timer loop when the second key has been pressed.

File size calculation program

The function FSIZE calculates the size of a named file and can be used either with the ProPascal program in chapter 9 or in your own assembler programs.

Its input parameter is the address of an FCB and this is passed on the stack. FSIZE will return to you in the register pair HL either the size of the file in sectors or −1 (if the file does not exist).

Disc free space program

The Pascal program provided uses the assembler procedures GET_DPB and GET_MAP and prints the amount of free space on a specified disc. GET_DPB and GET_MAP have been written so as to be of use in your own assembler programs and are suitably annotated.

GET_DPB returns a record containing the DPB for the specified drive. GET_MAP returns the allocation map for this drive. Both procedures expect two parameters: the drive code and an address of memory into which they will place the information required.

```
SIMPLE REACTION TIMER          RML Z80 Ass V 4.1 k    20-Mar-84    Page 1


                0001 *H SIMPLE REACTION TIMER
                0002 COM
                0003 *LM OFF
                0004
0005 =          0005 BDOSE                  EQU     5
0100 =          0006 TPA                    EQU     100H
                0007
0009 =          0008 FN_PRINT_STRING        EQU     9
0001 =          0009 FN_CONSOLE_IN          EQU     1
000B =          0010 FN_CONSOLE_STATUS      EQU     11
0000 =          0011 FN_SYSTEM_RESET        EQU     0
                0012
0024 =          0013 END_OF_STRING          EQU     '$'
000D =          0014 CR                     EQU     0DH
000A =          0015 LF                     EQU     0AH
0020 =          0016 SPACE                  EQU     20H
0001 =          0017 KEY_PRESSED            EQU     1
                0018
0000 =          0019 DELAY_2_MILLISECONDS   EQU     0.
                0020
                0021 MACRO   BDOS    P1
                0022                 LD      C,P1
                0023                 CALL    BDOSE
                0024 ENDM
                0025
                0026
0100            0027         ORG     TPA
0100            0028 START:
0100 31F502     0029         LD      SP,MY_TOP_STACK
                0030
0103 117C01     0031         LD      DE,SIGNON_MSG
                0032         BDOS    FN_PRINT_STRING
                0036
010B            0037 WAIT_FOR_START_KEY:
                0038         BDOS    FN_CONSOLE_IN                  ; WAIT FOR START KEY
0110 FE20       0042         CP      SPACE                          ; IS IT <SPACE> ?
0112 20F7       0043         JR      NZ,WAIT_FOR_START_KEY
                0044
0114 11B901     0045         LD      DE,STOP_MSG
                0046         BDOS    FN_PRINT_STRING
                0050
011C 3E00       0051         LD      A,0                            ; CLEAR TIMER COUNT
011E 32F401     0052         LD      (TICKS),A
0121 32F301     0053         LD      (TOCKS),A
                0054
0124            0055 TIMER_LOOP:
                0056         BDOS    FN_CONSOLE_STATUS              ; TEST IF A KEY HAS BE
0129 FE01       0060         CP      KEY_PRESSED                                   PRESS
012B 281A       0061         JR      Z,STOP_THE_CLOCK
                0062
012D            0063 SHORT_DELAY:
012D 0600       0064         LD      B,DELAY_2_MILLISECONDS         ; SHORT DELAY LOOP
                0065 1$:
012F E3         0066         EX      (SP),HL                        ; WASTE TIME
0130 E3         0067         EX      (SP),HL
```

```
SIMPLE REACTION TIMER          RML Z80 Ass V 4.1 k    20-Mar-84    Page 2


 0131 10FC     0068          DJNZ    1$
               0069
 0133 3AF401   0070          LD      A,(TICKS)                    ; INCREMENT THE TIME
 0136 C601     0071          ADD     A,1
 0138 27       0072          DAA
 0139 32F401   0073          LD      (TICKS),A
 013C 3AF301   0074          LD      A,(TOCKS)
 013F CE00     0075          ADC     A,0
 0141 27       0076          DAA
 0142 32F301   0077          LD      (TOCKS),A
               0078
 0145 18DD     0079          JR      TIMER_LOOP
               0080
 0147          0081 STOP_THE_CLOCK:
 0147 21F301   0082          LD      HL,TOCKS
               0083
 014A 3E00     0084          LD      A,0
 014C ED6F     0085          RLD                                  ; GET TOP NYBBLE INTO A
 014E C630     0086          ADD     A,'0'                        ; CONVERT TO A NUMBER.
 0150 32EE01   0087          LD      (TIME_MSG.4TH_DIGIT),A
               0088
 0153 3E00     0089          LD      A,0
 0155 ED6F     0090          RLD
 0157 C630     0091          ADD     A,'0'
 0159 32EF01   0092          LD      (TIME_MSG.3RD_DIGIT),A
               0093
 015C 23       0094          INC     HL
 015D 3E00     0095          LD      A,0
 015F ED6F     0096          RLD
 0161 C630     0097          ADD     A,'0'
 0163 32F001   0098          LD      (TIME_MSG.2ND_DIGIT),A
               0099
 0166 3E00     0100          LD      A,0
 0168 ED6F     0101          RLD
 016A C630     0102          ADD     A,'0'
 016C 32F101   0103          LD      (TIME_MSG.1ST_DIGIT),A
               0104
 016F 11D801   0105          LD      DE,TIME_MSG
               0106          BDOS    FN_PRINT_STRING
               0110
 0177          0111 LE_FINI:
               0112          BDOS    FN_SYSTEM_RESET
               0116
               0117 ;***************
               0118
```

```
DATA AREAS                    RML Z80 Ass V 4.1 k    20-Mar-84    Page 3


                0119 *H DATA AREAS
                0120
  017C          0121 SIGNON_MSG:
  017C 0D0A     0122                DEFB    CR,LF
  017E 52454143 0123                DEFM    'REACTION TIMER PROGRAM'
  0194 0D0A0D0A 0124                DEFB    CR,LF,CR,LF
  0198 50524553 0125                DEFM    'PRESS <SPACE> TO START THE TIMER'
  01B8 24       0126 END_OF_STRING
                0127
  01B9          0128 STOP_MSG:
  01B9 0D0A     0129                DEFB    CR,LF
  01BB 50524553 0130                DEFM    'PRESS <SPACE> TO STOP THE TIMER'
  01DA 24       0131 END_OF_STRING
                0132
  01DB          0133 TIME_MSG:
  01DB 0D0A0D0A 0134                DEFB    CR,LF,CR,LF
  01DF 54494D45 0135                DEFM    'TIME TAKEN IS:
  01EE          0136          TIME_MSG.4TH_DIGIT:
  01EE 00       0137                DEFB    0
  01EF          0138          TIME_MSG.3RD_DIGIT:
  01EF 00       0139                DEFB    0
  01F0          0140          TIME_MSG.2ND_DIGIT:
  01F0 00       0141                DEFB    0
  01F1          0142          TIME_MSG.1ST_DIGIT:
  01F1 00       0143                DEFB    0
  01F2 24       0144 END_OF_STRING
                0145
  01F3          0146 TOCKS:         DEFS    1
  01F4          0147 TICKS:         DEFS    1
                0148
  01F5          0149 STACK:
  01F5          0150                DEFS    256.
  02F5          0151 MY_TOP_STACK:
                0152
  0000          0153 END

Symbols:

0005 BDOSE              0000 CR                 0000 DELAY_2_MILLISECO  0024 END_OF_STRING
0001 FN_CONSOLE_IN      0008 FN_CONSOLE_STATUS  0009 FN_PRINT_STRING    0000 FN_SYSTEM_RESET
0001 KEY_PRESSED        0177 LE_FINI            000A LF                 02F5 MY_TOP_STACK
012D SHORT_DELAY        017C SIGNON_MSG         0020 SPACE              01F5 STACK
0100 START              0189 STOP_MSG           0147 STOP_THE_CLOCK     01F4 TICKS
0124 TIMER_LOOP         01DB TIME_MSG           01F1 TIME_MSG.1ST_DIGI  01F0 TIME_MSG.2ND_DIGI
01EF TIME_MSG.3RD_DIGI  01EE TIME_MSG.4TH_DIGI  01F3 TOCKS              0100 TPA
0108 WAIT_FOR_START_KE

No errors detected
```

```
FSIZE function for Propas        RML Z80 Ass V 4.1 k    20-Mar-84    Page 1


                0001 *H FSIZE function for Propas
                0002
 0011 =         0003 fn_search_first         = 11h
 0023 =         0004 fn_compute_file_size    = 23h
 0021 =         0005 fcb.random_record_offset= 33.
 0005 =         0006 BDOS                    = 5
 FFFF =         0007 break                   = -1
                0008
                0009 global  FSIZE
                0010
 0000           0011 FSIZE:
                0012 ; preserve environment
                0013 ; get parameters
                0014
                0015 ;break
 0000 E1        0016         POP     HL              ; return addr
 0001 223A00    0017         LD      (RETURN_ADDR),HL
 0004 D1        0018         POP     DE              ; addr of FCB
 0005 ED533C00  0019         LD      (fcb_addr),DE
                0020
                0021 ; check to see that the file exists
 0009 0E11      0022         LD      C,fn_search_first
 000B CD0500    0023         CALL    BDOS
 000E 3C        0024         INC     A               ; see if it exists
 000F 281D      0025         JR      Z,no_such_file
                0026
                0027 ; ok so lets find how big it is.
                0028
 0011 ED5B3C00  0029         LD      DE,(fcb_addr)
 0015 0E23      0030         LD      C,fn_compute_file_size
 0017 CD0500    0031         CALL    BDOS
                0032
 001A 2A3C00    0033         LD      HL,(fcb_addr)
 001D 112100    0034         LD      DE,fcb.random_record_offset
 0020 19        0035         ADD     HL,DE
 0021 7E        0036         LD      A,(HL)                  ; get low byte
 0022 4F        0037         LD      C,A
 0023 23        0038         INC     HL                      ; get middle byte
 0024 7E        0039         LD      A,(HL)
 0025 47        0040         LD      B,A
 0026 23        0041         INC     HL                      ; get low byte
 0027 7E        0042         LD      A,(HL)
 0028 5F        0043         LD      E,A
 0029 AF        0044         XOR     A
 002A 57        0045         LD      D,A
 002B EB        0046         ex      de,hl
 002C 1806      0047         JR      exit
                0048
 002E           0049 no_such_file:
 002E 3EFF      0050         LD      A,-1
 0030 67        0051         LD      H,A
 0031 6F        0052         LD      L,A
 0032 47        0053         LD      B,A
 0033 4F        0054         LD      C,A
                0055
```

```
FSIZE function for Propas        RML Z80 Ass V 4.1 k    20-Mar-84    Page 2


  0034           0056 exit:
  0034 ED5B3A00  0057         LD      DE,(return_addr)
  0038 D5        0058         PUSH    DE
  0039 C9        0059         ret
                 0060
  003A           0061 return_addr:
  003A           0062         defs    2
                 0063
  003C           0064 fcb_addr:
  003C           0065         defs    2
                 0066
  0000           0067 end


Symbols:

0005  BDOS               FFFF  BREAK              0034' EXIT               0021  FCB.RANDOM_RECORD
003C' FCB_ADDR           0023  FN_COMPUTE_FILE_S  0011  FN_SEARCH_FIRST    0000G FSIZE
002E' NO_SUCH_FILE       003A' RETURN_ADDR

No errors detected
```

```
program free_disc_space;
(*
  Program to calculate and display the amount of free space
  on a disc.
*)

const
   max_no_blocks_by_8   = 4095;  (* 32k/8 -1 *)

type
        byte    = 0..255;
        word    = 0..65535;
        dpb     = record
                    no_sectors_a_track  : word;
                    block_shift_factor  : byte;
                    block_mask          : byte;
                    extent_mask         : byte;
                    size_in_blocks      : word;
                    no_dir_entries      : word;
                    reserved            : array[0..3] of byte;
                    no_reserved_tracks  : word;
                  end;

   alloc_map    = array[0..max_no_blocks_by_8] of byte;

var
   free         : integer;
   drive        : byte;
   drive_name   : char;
   my_dpb       : dpb;
   an_alloc_map : alloc_map;

function free_disc_space(a_drive : byte) : integer;
(*
   find free disc space in sectors.
 *)

const
  unallocated   = 0;

var
  block_no      : integer;
  free_space    : word;

  procedure get_map(a_drive : byte; no_blocks_on_disc_div_8 : word;
                    var buffer_for_alloc_map : alloc_map);
  external;

  Procedure GET_DPB(drive : byte; var a_dpb : dpb);
  external;

  function check_block(block_number : integer;
                       var an_alloc_map : alloc_map) : byte;

  (*
     This function tests if a block is allocated or not,
     if it is it returns a '1', otherwise it returns a '0'
     for free block.
  *)
```

```
  var
     a_bit,this_bit     : byte;
     pwr2               : word;

  begin
    pwr2 := 1;
    this_bit := 0;

(*
   calculate bit no from block no (7-BLOCK_NO/8)
*)
    a_bit := 7 - (block_number mod 8);
    while this_bit < a_bit do
     begin
        pwr2 := pwr2*2;
        this_bit := this_bit +1;
      end;

(*    Return the answer  '0' - free, '1' - used.*)

    check_block := ((an_alloc_map[block_number div 8])
                    div pwr2 ) mod 2;
  end;

begin
  free_space := 0;
  get_dpb(a_drive,my_dpb);      (* get disc characteristics *)

 (* get allocation map *)
  get_map(a_drive,(my_dpb.size_in_blocks+8) div 8,an_alloc_map);

  for block_no := 0 to my_dpb.size_in_blocks do
   (* scan allocation map & count free space *)
    if check_block(block_no,an_alloc_map)= unallocated
      then free_space := free_space + 1;
  free_disc_space := (free_space * (my_dpb.block_mask+1)) div 8;
  (* free space in k bytes *)
end;

begin
  writeln('Simple Disc free space program              v1.2a');
  writeln;
  write('Enter drive name (A..P) ?');
  readln(drive_name);
  drive := ord(drive_name) - ord('A');

  free := free_disc_space(drive);       (* find free space *)

  write('Free space on disc ');
  write(drive_name,': is ',free,' kbytes from a total of ');
  write(((my_dpb.size_in_blocks+1)*(my_dpb.block_mask+1)) div 8);
  writeln(' kbytes.');
end.
```

```
Propas Proc Get_DPB              RML Z80 Ass V 4.1 k    20-Mar-84    Page 1


                0001 *H Propas Proc Get_DPB
                0002
                0003 ;***************************************************************************
                0004 ;GET_DPB:
                0005 ;--------
                0006 ;       Procedure to return a record containing the DPB for the specified
                0007 ;drive. Note that in order to ensure that the DPB is correct, the procedure
                0008 ;first performs a GET_CURRENT_DISC, saves the current drive code then resets
                0009 ;the specified drive.  It then selects the specified drive and performs a
                0010 ;BDOS FN_GET_DPB.  On exit it restore the current drive.
                0011 ;
                0012 ;type
                0013 ;       byte    = 0..255;
                0014 ;       word    = 0..65535;
                0015 ;       dpb     = record
                0016 ;                   no_sectors_a_track  : word;
                0017 ;                   block_shift_factor  : byte;
                0018 ;                   block_mask          : byte;
                0019 ;                   extent_mask         : byte;
                0020 ;                   size_in_blocks      : word;
                0021 ;                   no_dir_entries      : word;
                0022 ;                   reserved            : array[0..3] of byte;
                0023 ;                   no_reserved_tracks  : word;
                0024 ;                 end;
                0025 ;
                0026 ;Procedure GET_DPB(drive : byte; var a_dpb : dpb);
                0027 ;external;
                0028 ;
                0029 ;***************************************************************************
                0030
0005 =          0031 bdos                    = 5
001F =          0032 fn_get_dpb              = 1fh
0025 =          0033 fn_reset_drive          = 25h
0019 =          0034 fn_get_current_disc     = 19h
000E =          0035 fn_seldisc              = 0eh
                0036
000F =          0037 size_dpb                = 15.
                0038
                0039 global getdpb
                0040
0000            0041 getdpb:
0000            0042 get_dpb:
0000 D1         0043         pop     de                      ; get return addr
0001 ED534400   0044         ld      (return_addr),de
0005 E1         0045         pop     hl                      ; get addr a_dpb to return ans.
0006 224800     0046         ld      (var_a_dpb),hl
0009 F1         0047         pop     af                      ; get drive code
000A 324700     0048         ld      (drive),a
                0049
000D CD5C00     0050         call    bdos_get_current_disc   ; save current drive.
0010 324600     0051         ld      (current_disc),a
                0052
0013 110000     0053         ld      de,0
0016 3A4700     0054         ld      a,(drive)               ; calculate drive to reset
0019 3C         0055         inc     a
```

```
Propas Proc Get_DPB             RML Z80 Ass V 4.1 k    20-Mar-84    Page 2


001A 47         0056          ld      b,a
001B 37         0057          scf
001C            0058 loop:
001C CB13       0059          rl      e
001E CB12       0060          rl      d
0020 10FA       0061          djnz    loop
                0062
0022 CD5000     0063          call    bdos_reset_drive        ; reset & re-select specified drive
                0064
0025 3A4700     0065          ld      a,(drive)
0028 5F         0066          ld      e,a
0029 CD4A00     0067          call    bdos_seldisc
                0068
002C CD5600     0069          call    bdos_get_dpb            ; get dpb & copy data into a_dpb
002F ED5B4800   0070          ld      de,(var_a_dpb)
0033 010F00     0071          ld      bc,size_dpb
0036 EDB0       0072          ldir
                0073
0038 3A4600     0074          ld      a,(current_disc)
003B 5F         0075          ld      e,a
003C CD4A00     0076          call    bdos_seldisc            ; restore original drive.
                0077
003F 2A4400     0078          ld      hl,(return_addr)
0042 E5         0079          push    hl
0043 C9         0080          ret
                0081
                0082 ; data areas
                0083
0044            0084 return_addr:
0044            0085          defs    2
                0086
0046            0087 current_disc:
0046            0088          defs    1
                0089
0047            0090 drive:
0047            0091          defs    1
                0092
0048            0093 var_a_dpb:
0048            0094          defs    2
                0095
                0096 ;------------
                0097
                0098 ; declare bdos functions
                0099
004A            0100 bdos_seldisc:
004A 0E0E       0101          ld      c,fn_seldisc
004C CD0500     0102          call    bdos
004F C9         0103          ret
                0104
0050            0105 bdos_reset_drive:
0050 0E25       0106          ld      c,fn_reset_drive
0052 CD0500     0107          call    bdos
0055 C9         0108          ret
                0109
0056            0110 bdos_get_dpb:
```

```
Propas Proc Get_DPB              RML Z80 Ass V 4.1 k    20-Mar-84    Page 3


  0056 0E1F     0111          ld      c,fn_get_dpb
  0058 CD0500   0112          call    bdos
  005B C9       0113          ret
                0114
  005C          0115 bdos_get_current_disc:
  005C 0E19     0116          ld      c,fn_get_current_disc
  005E CD0500   0117          call    bdos
  0061 C9       0118          ret
                0119
  0000          0120 end


Symbols:

0005  BDOS                005C' BDOS_GET_CURRENT_   0056' BDOS_GET_DPB        0050' BDOS_RESET_DRIVE
004A' BDOS_SELDISC        0046' CURRENT_DISC        0047' DRIVE               0019  FN_GET_CURRENT_DI
001F  FN_GET_DPB          0025  FN_RESET_DRIVE      000E  FN_SELDISC          0000G GETDPB
0000' GET_DPB             001C' LOOP                0044' RETURN_ADDR         000F  SIZE_DPB
0048' VAR_A_DPB

No errors detected
```

```
Propas Proc GET_MAP             RML Z80 Ass V 4.1 k    20-Mar-84    Page 1


                0001 *h Propas Proc GET_MAP
                0002
                0003 ;*****************************************************************************
                0004 ; GET_MAP
                0005 ; -------
                0006 ;       Returns the allocation map for the specified drive. A call to GET_DPB
                0007 ; should have been called prior to this to a) get the disc size, b) reset the
                0008 ; drive.
                0009 ;
                0010 ;type
                0011 ;       byte    = 0..255;
                0012 ;       word    = 0..65535;
                0013 ;       dpb     = record
                0014 ;                   no_sectors_a_track  : word;
                0015 ;                   block_shift_factor  : byte;
                0016 ;                   block_mask          : byte;
                0017 ;                   extent_mask         : byte;
                0018 ;                   size_in_blocks      : word;
                0019 ;                   no_dir_entries      : word;
                0020 ;                   reserved            : array[0..3] of byte;
                0021 ;                   no_reserved_tracks  : word;
                0022 ;                 end;
                0023 ;
                0024 ;   alloc_map   = array[0..max_no_blocks_by_8] of byte;
                0025 ;
                0026 ;
                0027 ;  procedure get_map(a_drive : byte; no_blocks_on_disc_mod_8 : word;
                0028 ;                          var buffer_for_alloc_map : alloc_map);
                0029 ;  external;
                0030 ;
                0031 ;*****************************************************************************
                0032
0005 =          0033 bdos                    = 5
000E =          0034 fn_seldisc              = 0eh
0019 =          0035 fn_get_current_disc     = 19h
001B =          0036 fn_get_alloc_map        = 1bh
                0037
                0038 global  getmap
                0039
0000            0040 getmap:
0000            0041 get_map:
0000 D1         0042         pop     de                      ; get return addr
0001 ED533700   0043         ld      (return_addr),de
0005 E1         0044         pop     hl                      ; var alloc_map
0006 223D00     0045         ld      (var_alloc_map),hl
0009 E1         0046         pop     hl                      ; no bytes in alloc map
000A 223B00     0047         ld      (map_size),hl
000D F1         0048         pop     af                      ; get drive in a
000E 323900     0049         ld      (drive),a
                0050
0011 CD4500     0051         call    bdos_get_current_disc   ; save current drive
0014 323A00     0052         ld      (current_disc),a
                0053
0017 3A3900     0054         ld      a,(drive)               ; select specified drive
001A 5F         0055         ld      e,a
```

```
Propas Proc GET_MAP              RML Z80 Ass V 4.1 k    20-Mar-84    Page 2


001B CD3F00   0056          call    bdos_seldisc
              0057
001E CD4B00   0058          call    bdos_get_alloc_map      ; get addr alloc map
0021 ED5B3D00 0059          ld      de,(var_alloc_map)      ; copy contents of alloc map
0025 ED4B3B00 0060          ld      bc,(map_size)           ; to BUFFER_FOR_ALLOC_MAP
0029 EDB0     0061          ldir
              0062
002B 3A3A00   0063          ld      a,(current_disc)        ; restore current drive.
002E 5F       0064          ld      e,a
002F CD3F00   0065          call    bdos_seldisc
              0066
0032 2A3700   0067          ld      hl,(return_addr)
0035 E5       0068          push    hl
0036 C9       0069          ret
              0070
              0071 ;----------
              0072
              0073 ; data areas
              0074
0037          0075 return_addr:
0037          0076          defs    2
              0077
0039          0078 drive:
0039          0079          defs    1
              0080
003A          0081 current_disc:
003A          0082          defs    1
              0083
003B          0084 map_size:
003B          0085          defs    2
              0086
003D          0087 var_alloc_map:
003D          0088          defs    2
              0089
              0090
              0091 ;-------------
              0092
              0093 ; bdos routines
              0094
003F          0095 bdos_seldisc:
003F 0E0E     0096          ld      c,fn_seldisc
0041 CD0500   0097          call    bdos
0044 C9       0098          ret
              0099
0045          0100 bdos_get_current_disc:
0045 0E19     0101          ld      c,fn_get_current_disc
0047 CD0500   0102          call    bdos
004A C9       0103          ret
              0104
004B          0105 bdos_get_alloc_map:
004B 0E1B     0106          ld      c,fn_get_alloc_map
004D CD0500   0107          call    bdos
0050 C9       0108          ret
              0109
              0110
```

```
Propas Proc GET_MAP              RML Z80 Ass V 4.1 k    20-Mar-84    Page 3

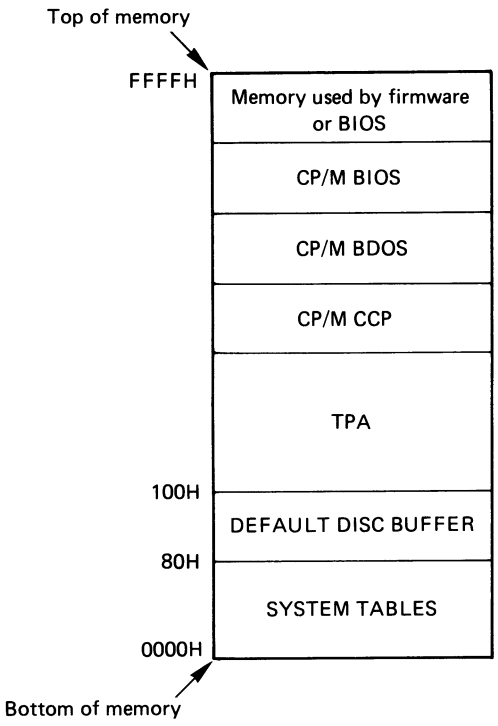

Symbols:

0005  BDOS                004B' BDOS_GET_ALLOC_MA    0045' BDOS_GET_CURRENT_    003F' BDOS_SELDISC
003A' CURRENT_DISC         0039' DRIVE                001B  FN_GET_ALLOC_MAP     0019  FN_GET_CURRENT_DI
000E  FN_SELDISC           0000G GETMAP               0000' GET_MAP              003B' MAP_SIZE
0037' RETURN_ADDR          003D' VAR_ALLOC_MAP

No errors detected
```

Appendix B: CP/M Memory Map

The organisation of memory under CP/M is shown in the diagram below. Your programs can be placed in the Transient Program Area (TPA).



Appendix C: ASCII Code Tables

The first table below shows the standard ASCII 7-bit character set. In the second table, we have broken this down to show the decimal value of each key as well as the hexadecimal value. Where relevant, the CTRL code for a character is also shown; this shows how you can generate the character by pressing a combination of the CTRL key and another key.

Most of the actions in the first table are machine-independent; however, some of them are not. You should thus check the implementation on your system and update the second table accordingly.

	0	1	2	3	4	5	6	7
0	NUL	DLE		0	@	P	‾	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	£	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	´	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	¦
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	↑	n	-
F	SI	US	/	?	O	#	o	DEL

Dec	Hex	Name	Effect	Key Stroke	Dec	Hex	Name	Effect
0	00H	NUL	Do nothing	CTRL/@	64	40H		@
1	01H	SOH	Start of header	CTRL/A	65	41H		A
2	02H	STX	Start of text	CTRL/B	66	42H		B
3	03H	ETX	End of text	CTRL/C	67	43H		C
4	04H	EOT	End of transmission	CTRL/D	68	44H		D
5	05H	ENQ	Enquiry	CTRL/E	69	45H		E
6	06H	ACK	Acknowledge	CTRL/F	70	46H		F
7	07H	BEL	Bell	CTRL/G	71	47H		G
8	08H	BS	Backspace	CTRL/H	72	48H		H
9	09H	HT	Horizontal tab	CTRL/I	73	49H		I
10	0AH	LF	Linefeed	CTRL/J	74	4AH		J
11	0BH	VT	Vertical tab	CTRL/K	75	4BH		K
12	0CH	FF	Form feed	CTRL/L	76	4CH		L
13	0DH	CR	Carriage return	CTRL/M	77	4DH		M
14	0EH	SO	Shift out	CTRL/N	78	4EH		N
15	0FH	SI	Shift in	CTRL/O	79	4FH		O
16	10H	DLE	Data Link Escape	CTRL/P	80	50H		P
17	11H	DC1	XON, Transmission ON	CTRL/Q	81	51H		Q
18	12H	DC2	XON, Transmission ON	CTRL/R	82	52H		R
19	13H	DC3	XOFF, Transmission OFF	CTRL/S	83	53H		S
20	14H	DC4	XOFF,Transmission OFF	CTRL/T	84	54H		T
21	15H	NAK	Negative Acknowledge	CTRL/U	85	55H		U
22	16H	SYN	Synchronous Idle	CTRL/V	86	56H		V
23	17H	ETB	End of Transmission Block	CTRL/W	87	57H		W
24	18H	CAN	Cancel	CTRL/X	88	58H		X
25	19H	EM	End of medium	CTRL/Y	89	59H		Y
26	1AH	SUB	Substitute character	CTRL/Z	90	5AH		Z
27	1BH	ESC	Escape key	CTRL/[	91	5BH		[
28	1CH	FS	File separator	CTRL/\	92	5CH		\
29	1DH	GS	Group separator	CTRL/]	93	5DH		]
30	1EH	RS	Record separator	CTRL/^	94	5EH		^
31	1FH	US	Unit separator	CTRL/_	95	5FH		_
32	20H		Space		96	60H		`
33	21H		!		97	61H		a
34	22H		"		98	62H		b
35	23H		£		99	63H		c
36	24H		$		100	64H		d
37	25H		%		101	65H		e
38	26H		&		102	66H		f
39	27H		'		103	67H		g
40	28H		(		104	68H		h
41	29H		)		105	69H		i
42	2AH		*		106	6AH		j
43	2BH		+		107	6BH		k
44	2CH		,		108	6CH		l
45	2DH		-		109	6DH		m
46	2EH		.		110	6EH		n
47	2FH		/		111	6FH		o
48	30H		0		112	70H		p
49	31H		1		113	71H		q
50	32H		2		114	72H		r
51	33H		3		115	73H		s
52	34H		4		116	74H		t
53	35H		5		117	75H		u
54	36H		6		118	76H		v
55	37H		7		119	77H		w
56	38H		8		120	78H		x
57	39H		9		121	79H		y
58	3AH		:		122	7AH		z
59	3BH		;		123	7BH		{
60	3CH		<		124	7CH		|
61	3DH		=		125	7DH		}
62	3EH		>		126	7EH		~
63	3FH		?		127	7FH	DEL	Backspace and delete

Appendix D: BDOS Functions

This appendix starts with a summary of the BDOS-calling mechanism. The BDOS functions are then listed on this page and the next, first in numeric order and then in their appropriate groups.

The rest of the appendix contains detailed descriptions of all functions. The functions themselves are laid out in alphabetic order, with the function code in decimal at the top of each description and in hexadecimal at the bottom.

BDOS-calling mechanism

- Load C register with function code
- Load registers D and E if necessary
- Call BDOS at address 0005H
- Pick up any information returned by BDOS in registers A, H and L

Remember: the BDOS corrupts 99 per cent of household registers!

Summary of BDOS functions

Table D.1 BDOS functions in numeric order

Fn. No.	Function	Fn. No.	Function
0	System Reset	20	Read Sequential
1	Console Input	21	Write Sequential
2	Console Output	22	Create (Make) File
3	Reader Input	23	Rename File
4	Punch Output	24	Return Logged-in Drives
5	List Output	25	Return Current Disc
6	Direct Console I/O	26	Set DMA Address
7	Get I/O Byte	27	Get Allocation Address
8	Set I/O Byte	28	Write Protect Disc
9	Print String	29	Get Read-Only Indicators
10	Read Console Buffer	30	Set File Attributes
11	Get Console Status	31	Get Disc Parameters Address
12	Return Version Number	32	Set/Get User Code
13	Reset Disc System	33	Read Random
14	Select Disc	34	Write Random
15	Open File	35	Compute File Size
16	Close File	36	Set Random Sector
17	Search for First Entry	37	Reset Drive
18	Search for Next Entry	38/39	Reserved
19	Delete File	40	Write Random with Zero Fill

Table D.2 BDOS function groups

Console and simple I/O functions		*Disc functions*	
Console Input	1	Reset Disc System	13
Console Output	2	Select Disc	14
Reader Input	3	Return Logged-in Drives	24
Punch Output	4	Return Current Disc	25
List Output	5	Get Allocation Address	27
Direct Console I/O	6	Write Protect Disc	28
Get I/O Byte	7	Get Read-Only Indicators	29
Set I/O Byte	8	Get Disc Parameters Address	31
Print String	9	Set/Get User Code	32
Read Console Buffer	10	Reset Drive	37
Get Console Status	11		
File handling functions		*Miscellaneous functions*	
Open File	15	System Reset	0
Close File	16	Return Version Number	12
Search for First Entry	17		
Search for Next Entry	18		
Delete File	19		
Read Sequential	20		
Write Sequential	21		
Create (Make) File	22		
Rename File	23		
Set DMA Address	26		
Set File Attributes	30		
Read Random	33		
Write Random	34		
Compute File Size	35		
Set Random Sector	36		
Write Random with Zero Fill	40		

16 CLOSE FILE

Registers on entry	*Registers on exit*
C: 10H **DE:** FCB address	**A:** Directory code

Effect

This function is the reverse of Open File (function 15). It closes the file described by the FCB whose address is held in DE by writing the enclosed details in the appropriate disc directory. Wildcard characters (?) can be used in the filename.

The directory code returned is in the range 0 to 3 for a successful operation. If the filename cannot be found in the directory, FFH (255 decimal) is returned.

The FCB must have been used for an Open file or Create File function.

Example

```
BDOS                =   0005H
FN_CLOSE_FILE       =   16
DATAFILE_FCB:       ...

                    ...
                    ...
                    LD   DE,DATAFILE_FCB
                    LD   C,FN_CLOSE_FILE
                    CALL BDOS

                    INC  A
                    JN   Z,EXIT
FAILED_CLOSE:       ...             ;Identify failed file
                    ...             ;with message
EXIT:               ...
```

COMPUTE FILE SIZE 35

Registers on entry	*On exit*
C: 23H **DE:** FCB address	Bytes 33 to 35 of FCB set to block after end of file

Effect

This function allows you to calculate the size of a file. It returns the number of 128-byte sectors in the file by setting the record pointer in the FCB to the sector immediately after the end of the file. You can then easily append data to the end of the file.

The register pair DE must point to a 36-byte FCB for the file in question, and the FCB must not contain any wildcard characters (?).

On return, bytes 33 and 35 of the FCB contain the file size as a 16-bit value; in effect, this is the sector address of the sector following the end of the file. If byte 35 contains 1 on return, the file contains the maximum sector count (65536).

Compute File Size will always give you an answer whether or not the file exists. Thus, you should check that the file exists by using Search for First.

You can append data to the end of an existing file by merely calling Compute File Size to set the random record position to the end of the file, then performing a sequence of random writes starting at the sector address contained in the record pointer.

When you write a file sequentially, the number of sectors of data in the file corresponds to the number returned by Compute File Size. However, if you create the file in random mode and leave 'holes' in it, Compute File Size will give a larger number than the number of data sectors in the file. If, for example, you write only the last sector of an 8 Megabyte file in random mode (that is, sector 65535), the file size is 65536 but only one sector of data is allocated.

23H

35 COMPUTE FILE SIZE

Example

```
BDOS                    =    0005H
FN_COMPUTE_FILE_SIZE    =    35
DATAFILE_FCB      ...

                  ...
                  ...
                  LD   DE,DATAFILE_FCB
                  LD   C,FN_COMPUTE_FILE_SIZE
                  CALL BDOS

                  ...       ;File size is in bytes 33 & 34
                  ...       ;of DATAFILE_FCB.
```

CONSOLE INPUT 1

Registers on entry	*Registers on exit*
C: 01H	A: ASCII character

Effect

Reads a character from the keyboard into register A.

Printable characters, together with some of the special control characters (RETURN, linefeed and backspace) are echoed to the screen. Tab characters (CTRL/I) are expanded so as to move the cursor to the next tab position. Other control characters are not echoed to the screen.

Control is not returned to your program until a character has been typed in. Thus, function 1 is not suitable for real-time use. If you are writing a real-time program you should use Direct Console I/O (function 6) instead.

Example

```
BDOS            =    0005H
FN_CONSOLE_IN   =    1
                ...
                ...
                LD   C,FN_CONSOLE_IN
                CALL BDOS
                                ;ASCII character
                                 ;is in register A.
                ...
                ...
```

2 CONSOLE OUTPUT

Registers on entry	*Registers on exit*
C: 02H **E:** ASCII character	

Effect

The ASCII character in register E is sent to the screen. Tabs are expanded and checks are made for start/stop scrolling (CTRL/S) from the keyboard.

Example

```
BDOS            =   0005H
FN_CONSOLE_OUT  =   2
                ...
                ...
                LD  E,'C'
                LD  C,FN_CONSOLE_OUT
                CALL BDOS
                ...
                ...
```

CREATE (MAKE) FILE 22

Registers on entry	*Registers on exit*
C: 16H **DE:** FCB address	**A:** Directory code

Effect

Creates an entry in the directory for a file named in the FCB addressed by DE. This FCB must name a file that does not currently exist in the directory. You can make sure that the file does not exist by using a preceding Delete File function.

Create File also activates the FCB for file operations; thus, a subsequent Open File function is not needed.

After a successful operation, a value in the range 0 to 3 is returned in the A register. For an unsuccessful operation, the value FFH (255 decimal) is returned, indicating that the directory is full.

Note that if you attempt to create a file that already exists, you will corrupt the disc structure.

Example

```
BDOS                    =    0005H
FN_CREATE_FILE          =    22
DATAFILE_FCB:     ...

                  ...
                  ...
                  LD   DE,DATAFILE_FCB
                  LD   C,FN_CREATE_FILE
                  CALL BDOS

                  INC  A
                  JN   Z,PROCESS
FAILED_CREATION:  ...                    ;Directory full
                  ...
PROCESS:          ...
```

19 DELETE FILE

Registers on entry	*Registers on exit*
C: 13H **DE:** FCB address	**A:** Directory code

Effect

This function removes all files that match the FCB addressed by register pair DE. The filename and filetype positions can contain wildcard characters (?) but the drive code must be unambiguous.

If the file or files cannot be found, the value FFH (255 decimal) will be returned. Otherwise, a value in the range 0 to 3 will be returned.

Example

```
BDOS                =   0005H
FN_DELETE_FILE      =   19
DATAFILE_FCB:   ...

                ...
                ...
                LD   DE,DATAFILE_FCB
                LD   C,FN_DELETE_FILE
                CALL BDOS

                INC  A
                JN   Z,EXIT
FAILED_DELETION: ...                    ;Identify failed file
                ...                     ;with message
EXIT:           ...
```

DIRECT CONSOLE I/O 6

Registers on entry	*Registers on exit*
C: 06H E: 0FFH (input) char. (output)	A: ASCII character or status

Effect

This function allows you to display output on the screen or read from the keyboard. It bypasses BDOS control character functions such as CTRL/C, CTRL/S, CTRL/Q and CTRL/P.

Upon entry, register E should contain FFH (indicating that you want console input), FEH (indicating that you want the console status), or an ASCII character that you want to output.

When you put FFH in register E, Direct Console I/O returns zero in the A register if no character is ready, otherwise register A contains the next character input. Input is not echoed to the screen.

Versions of CP/M later than 2.2 accept FEH in register E as being a request for the keyboard status. They will return either 0 if no character is waiting, or 1 if a character has been input.

If the value in E is not FFH or FEH, CP/M assumes that it is a valid ASCII character and sends it to the screen. In this case, the contents of the A register are undefined.

You must not mix Direct Console I/O and other console I/O functions.

06H

6 DIRECT CONSOLE I/O

Example

The following code outputs the character Y to the screen using Direct Console I/O.

```
BDOS            =   0005H
FN_DIRECT_IO    =   6
                ...
                ...
                LD  E,'Y'           ;Load character to be output
                LD  C,FN_DIRECT_IO
                CALL BDOS
                ...
                ...
```

The following code uses Direct Console I/O to input the character CTRL/C without rebooting CP/M.

```
BDOS            =   0005H
FN_DIRECT_IO    =   6
CTRL_C          =   3
INPUT_MODE      =   0FFH
                ...
                ...
WAIT_FOR_CHAR:  LD  E,INPUT_MODE    ;Indicate input request
                LD  C,FN_DIRECT_IO
                CALL BDOS
                JR  Z,WAIT_FOR_CHAR ;Character input?

                CP  CTRL_C          ;Test for CTRL/C
                JP  Z,EXIT
PROCESS_CHAR:   ...
                ...
EXIT:           ...
                ...
```

GET ALLOCATION ADDRESS 27

Registers on entry	*Registers on exit*
C: 1BH	**HL:** Allocation address

Effect

This function returns the address in memory of the allocation map for the currently selected drive.

An allocation map is a record of which allocation blocks on a disc are in use and which ones are not. There is an allocation map for each disc and it consists of a bit map, each bit of which corresponds to an allocation block.

Note that the allocation information may be invalid if the disc selected has been marked as read-only.

Example

```
BDOS                    =    0005H
FN_GET_ALLOCATION_ADDRESS =  27
                  ...
                  ...
                  ...
                  LD   C,FN_GET_ALLOCATION_ADDRESS
                  CALL BDOS
                  ...         ;Allocation map address is in
                  ...         ;register pair HL
                  ...
```

1BH

11 GET CONSOLE STATUS

Registers on entry	*Registers on exit*
C: 0BH	A: Status information

Effect

This function allows you to test if a character has been input at the keyboard but to continue processing if it has not.

If a character has been typed, the function returns the value 1; otherwise, 0 is returned.

Example

```
TIMER_LOOP:       LD   C,FN_CONSOLE_STATUS   ;Test if a key has
                                             ;been pressed.
                  CALL BDOS

                  OR   A                     ;Set flags:
                                             ;  - Z  no key pressed
                                             ;  - NZ key pressed.
                  JR   NZ,STOP_THE_CLOCK
                  CALL INCREMENT_THE_CLOCK
                  JR   TIMER_LOOP

STOP_THE_CLOCK:   ...
                  ...
```

GET DISC PARAMETERS ADDRESS 31

Registers on entry	*Registers on exit*
C: 1FH	**HL:** DPB address

Effect

This function returns the address of the BIOS Disc Parameter Block (DPB) in register pair HL. Full details of the DPB contents are given in chapter 6.

Example

```
BDOS                    =    0005H
FN_GET_DPB_ADDRESS      =    31
                ...
                ...
                ...
                LD   C,FN_GET_DPB_ADDRESS
                CALL BDOS

                ...          ;DPB address is in register pair HL
                ...
```

7 GET I/O BYTE

Registers on entry	*Registers on exit*
C: 07H	A: I/O byte value

Effect

This function returns the current value of IOBYTE in register A.

Example

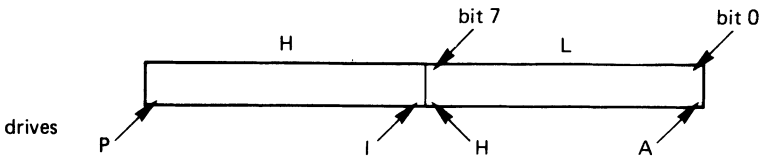
```
BDOS            =   0005H
FN_GET_IO_BYTE  =   7
                ...
                ...
                LD  C,FN_GET_IO_BYTE
                CALL BDOS
                ...         ;I/O byte value is in register A.
                ...
```

GET READ-ONLY INDICATORS 29

Registers on entry	*Registers on exit*
C: 1DH	**HL:** Status info.

Effect

This function returns a value in register pair HL where each bit indicates the status of a specific drive using the arrangement



If a bit is set, the related drive has been write-protected using Write Protect Disc (function 28).

Example

```
BDOS                        =    0005H
FN_GET_READ_ONLY_INDICATORS =    25
                ...
                ...
                LD   C,FN_GET_READ_ONLY_INDICATORS
                CALL BDOS

                LD  A,L
                AND 1
                JR  Z,PROCESS   ;Jump if drive A is not
                                ;write protected

UNPROTECT:      ...              ;Remove write protection
                ...

PROCESS:        ...
```

1DH

5 LIST OUTPUT

Registers on entry	*Registers on exit*
C: 05H **E:** ASCII character	

Effect

The ASCII character in register E is sent to the logical listing device. Tabs are expanded and checks are made for other CTRL codes.

Example

```
BDOS            =   0005H
FN_LIST_OUT     =   5
                ...
                ...
                LD  E,'C'
                LD  C,FN_LIST_OUT
                CALL BDOS
                ...
                ...
```

OPEN FILE 15

Registers on entry	*Registers on exit*
C: 0FH **DE:** FCB address	**A:** Directory code

Effect

This function opens a file whose entry exists in the disc directory with the currently active user number. You should not try to access a file unless it has first been successfully opened.

CP/M scans the referenced disc directory for a match in fields 1 to 14 of the FCB referenced by DE. If you put a wildcard (?) character in any position of the filename, any directory character will be accepted. Normally, you should not do this.

If an entry is found in the directory, the relevant directory information will be copied into bytes 16 to 31 of the FCB. You can then access the file with read and write operations.

If the file cannot be found, Open File returns the value FFH (255 decimal) in register A. If the file is found, a directory code in the range 0 to 3 is returned. You can use the directory code to find the directory entry in the current DMA buffer of your program. If you multiply the contents of the A register by 32 (shift the A register left by 5 bits) this will give you the start address of the directory entry in the buffer.

Note that the current record pointer in the FCB (byte 32) must be zeroed by your program if the file is to be accessed sequentially from the first record.

Example

```
BDOS                =   0005H
FN_OPEN_FILE        =   15
DATAFILE_FCB:   ...

                ...
                ...
                LD   DE,DATAFILE_FCB
                LD   C,FN_OPEN_FILE
                CALL BDOS

                INC  A
                JN   Z,EXIT
FAILED_OPEN:    ...             ;Identify failed file
                ...             ;with message
EXIT:           ...
```

0FH

9 PRINT STRING

Registers on entry	*Registers on exit*
C: 09H **DE:** String address	

Effect

Sends to the screen a character string which is stored in memory at the address held in the register pair DE. The end of the string should be marked by a $ (dollar) character as shown below.



Tab characters (CTRL/I) are expanded so as to move the cursor to the next tab position. A check is also made for start/stop scrolling characters (CTRL/S) and printer echo (CTRL/P) on keyboard input.

This function will not normally print '$' characters (except under certain versions of CP/M and under certain conditions – see chapter 7). To print '$' characters, you must use Console Out (function 2).

Example

```
BDOS            =   0005H
FN_PRINT_STRING =   9
END_OF_STRING   =   '$'

MESSAGE:        DEFM 'This is a string of text'
                DEFB END_OF_STRING
                ...
                LD   DE,MESSAGE   ;Pick up message
                                  ;start address.

                LD   C,FN_PRINT_STRING
                CALL BDOS         ;Print message.
                ...
```

PUNCH OUTPUT 4

Registers on entry	*Registers on exit*
C: 04H **E:** ASCII character	**A:** ASCII character

Effect

This function is a throwback to the days when input/output to CP/M was on paper tape. It sends the ASCII character in register E to the logical punch device.

Some programs such as PIP treat the device PUN as being special and precede/follow any file transfers to PUN: with a series of NULL (0H) characters.

Example

```
BDOS            =    0005H
FN_PUNCH_OUT    =    4
                ...
                ...
                LD   E,'C'
                LD   C,FN_PUNCH_OUT
                CALL BDOS
                ...
                ...
```

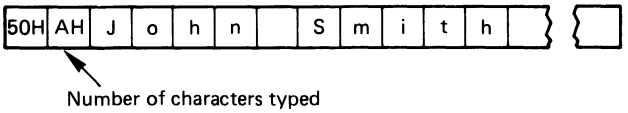
10 READ CONSOLE BUFFER

Registers on entry	*Registers on exit*
C: 0AH **DE:** Buffer address	

Effect

Reads a line input from the keyboard into a buffer whose address is in register pair DE. Keyboard input ends either when the buffer overflows or a carriage return or linefeed is typed.

The input buffer has the following form.



If you have not reserved storage space for an input buffer, the BDOS will use whatever memory the DE registers point to. You should therefore ensure that sufficient space is allocated.

The size of the buffer should be in the range 1 to 255 bytes and you need a region of memory equal to the buffer length plus 2 bytes. If the buffer length is marked as zero, the BDOS will treat it as having a size of 1 byte and will transfer 1 character.

A number of line-editing facilities are available during input. Their description and key combinations are given in the following table.

READ CONSOLE BUFFER 10

Key combination	*Effect*
RUB/DEL	Removes the last character and backspaces one character position
CTRL/C	Reboots the operating system when it is at the beginning of the line
CRTL/E	Forces end of line
CTRL/H	Removes the last character and backspaces one character position
CTRL/J	Terminates line input (LINEFEED)
CTRL/M	Terminates line input (RETURN)
CTRL/P	Printer echo on/off
CTRL/R	Retypes current line on next line, showing all the changes made
CTRL/U	Deletes current line
CTRL/X	Same as CTRL/U

Certain operations return the cursor to the leftmost position (for example, CTRL/U). In these cases, the cursor will be placed in the column position where the prompt ended, thus making operator data input and line correction more legible. In earlier releases of CP/M, the cursor returned to the extreme left margin.

Example

```
BDOS            =   0005H
FN_READ_BUFFER  =   10
                ...
BUFFER_LENGTH   =   80
BUFFER:         DEFS BUFFER_LENGTH
                DEFB BUFFER_LENGTH+1
                ...
                LD   DE,BUFFER      ;Set up buffer
                                    ;start address.
                LD   C,FN_READ_BUFFER
                CALL BDOS
                ...
```

33 READ RANDOM

Registers on entry	*Registers on exit*
C: 21H **DE:** FCB address	**A:** Return code

Effect

This function reads a 128-byte sector of data from a file into the current data buffer. The file must have been opened using Open File (function 15).

The sector number in the file is specified by the contents of bytes 33 to 35 of the FCB. Byte 33 is the least-significant byte and byte 35 the most-significant byte. Bytes 33 and 34 together form a 16-bit sector number in the range 0 to 65535. Byte 35 is used by CP/M to calculate the size of the file and you should set it to zero before using the function.

To use Read Random, you first set the sector number in bytes 33 and 34, set byte 35 to zero and then call the BDOS. Upon return, register A contains one of the error codes shown in table D.3 (on page 132) or zero, if the operation was successful. In the latter case, the current DMA address will hold the block required. The sector number in bytes 33 and 34 will not have been incremented, so a subsequent read will read the same record.

CP/M automatically sets the logical extent and current sector values in the FCB. Thus, you can read or write the file sequentially from the current randomly accessed position. If you do this, you should note that the last block that was randomly read will be reread when you switch to sequential reading, and the last record will be rewritten when you switch to sequential writing.

READ RANDOM 33

Example

```
BDOS                =   0005H
FN_READ_RANDOM      =   33
DATAFILE_FCB        ...

                    ...
                    ...
                    LD   (DATAFILE_FCB.RANDOM_BLOCK),HL ;Load random
                                                        ;block no
                                                        ;from HL
                    LD   DE,DATAFILE_FCB
                    LD   C,FN_READ_RANDOM
                    CALL BDOS
                    JP   Z,PROCESS              ;Read successful

ERROR:              ...
                    ...
PROCESS:            ...
```

Table D.3 File handling error codes

Error code	*Meaning when caused by read operation*	*Meaning when caused by write operation*
1	Reading unwritten data	–
2	–	Disc full
3	Cannot close current extent*	Cannot close current extent*
4	Seek to unwritten extent	–
5	–	New extent cannot be created (directory overflow)
6	File size overflow (byte 35 of FCB too big)	File size overflow

*Does not normally occur under proper system operation. If it does, the error can be cleared by re-reading extent zero (as long as the disc is not physically write-protected).

20 READ SEQUENTIAL

Registers on entry	*Registers on exit*
C: 14H **DE:** FCB address	**A:** Return code

Effect

Reads the next 128-byte sector of data from the file into memory at the current DMA address. The file is described by the FCB whose address is held in DE and this FCB must have been activated by an Open File or Create File function.

If the read is successful, a zero value will be returned in register A, otherwise a non-zero value will be returned. This could happen, for example, if end-of-file is detected.

The data sector is read from a position in the disc logical extent given by byte 32 of the FCB. This byte is then incremented by CP/M to point to the next sector. If byte 32 overflows, the next logical extent is opened and byte 32 reset to zero ready for the next read.

Example

```
BDOS                    =    0005H
FN_READ_SEQUENTIAL      =    20
DATAFILE_FCB:     ...

                  ...
                  ...
                  LD   DE,DATAFILE_FCB
                  LD   C,FN_READ_SEQUENTIAL
                  CALL BDOS

                  JP   Z,PROCESS
FAILED_READ:      ...                    ;End of file reached
                  ...
PROCESS:          ...
```

READER INPUT 3

Registers on entry	*Registers on exit*
C: 03H	A: ASCII character

Effect

This function is a throwback to the days when input/output was on paper tape. It reads the next ASCII character from the logical reader into register A.

Example

```
BDOS            =   0005H
FN_READER_IN    =   3
                ...
                ...
                LD  C,FN_READER_IN
                CALL BDOS
                                ;ASCII character
                                ;is in register A.
                ...
                ...
```

23 RENAME FILE

Registers on entry	*Registers on exit*
C: 17H **DE:** FCB address	**A:** Return code

Effect

Before calling this function, the FCB addressed by register pair DE is set up by you to hold two file descriptions. The first 16 bytes of the FCB contain a description of the old file, the next 16 bytes a description of the new file.

The drive code at byte 0 is used to select the drive; the drive code at byte 16 of the FCB is assumed to be zero.

After a successful operation, a value in the range 0 to 3 is returned in A. If the old filename cannot be found in the directory, FFH (255 decimal) is returned.

Note that this function will not accept the wildcard character (?) in the FCB. You must also ensure that the new file name does not already exist.

Example

```
BDOS                =    0005H
FN_RENAME_FILE      =    23
DATAFILE_FCB:    ...

                 ...
                 ...
                 LD   DE,DATAFILE_FCB
                 LD   C,FN_RENAME_FILE
                 CALL BDOS

                 INC  A
                 JN   Z,PROCESS
FAILED_RENAME:   ...              ;Old filename not in directory
                 ...
PROCESS:         ...
```

RESET DISC SYSTEM 13

Registers on entry	*Registers on exit*
C: 0DH	

Effect

Write Protect Disc (function 28) lets you protect a disc against writing from within your program. Reset Disc System restores the entire disc system to a reset state where all discs are set to read/write. Drive A will be the only currently selected drive and the default data buffer address will be set to 0080H.

Example

```
BDOS                    =    0005H
FN_RESET DISC_SYSTEM    =    13
                  ...
                  ...
                  LD   C,FN_RESET_DISC_SYSTEM
                  CALL BDOS

                  ...
                  ...
```

0DH

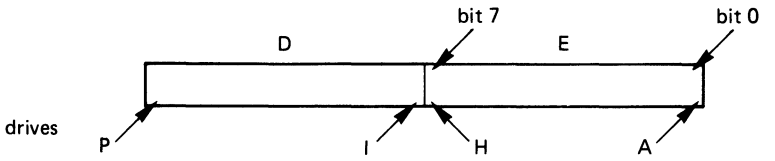
37 RESET DRIVE

Registers on entry	*Registers on exit*
C: 25H **DE:** Drive info	**A:** 00H

Effect

Resets one or more drives so that any write protection caused by a call to Write Protect Disc (function 28) is cancelled and the drives are returned to the Read/Write status.

Drives are specified by setting bits in the register pair DE using the following arrangement.



To maintain compatibility with MP/M, CP/M returns a zero value in register A.

Example

The following code resets drives A, B, C and D.

```
BDOS                =   0005H
FN_RESET_DRIVE      =   37
                ...

                ...
                ...
                LD   D,0                ;Specify drives in
                                        ;registers DE
                LD   E,15
                LD   C,FN_RESET_DRIVE
                CALL BDOS
```

RETURN CURRENT DISC 25

Registers on entry	*Registers on exit*
C: 19H	A: Current disc

Effect

This function returns the drive number of the currently selected drive. The drive numbers are in the range 0 to 15 and correspond to drives A to P.

Example

The following example checks if drive A is the currently selected drive and warns the operator if it is not.

```
BDOS                    =   0005H
FN_RETURN_CURRENT_DRIVE =   25
DRIVE_A                 =   0
                ...
                ...
                LD   C,FN_RETURN_CURRENT_DRIVE
                CALL BDOS

                CP   DRIVE_A
                JR   Z,PROCESS  ;Jump if drive A is currently
                                ;selected drive

PROMPT:         ...             ;Prompt operator to make disc
                ...             ;available and select drive by
                ...             ;program

PROCESS:        ...
                ...
```

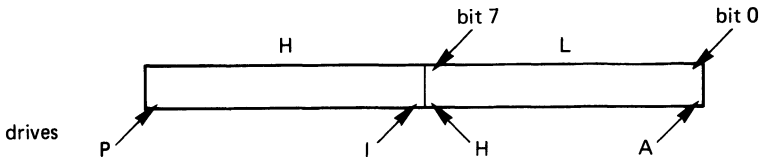
19H

24 RETURN LOGGED-IN DRIVES

Registers on entry	*Registers on exit*
C: 18H	**HL:** Drive status

Effect

This function returns a 16-bit value in the register pair HL, where each bit position corresponds to a drive using the following scheme.



If the appropriate bit is set, the drive is logged-in as a result of either of the following

- An explicit drive selection
- An implicit drive selection caused by a file operation that specified the drive field

Example

The following example checks if drive B is logged-in and warns the operator if it is not.

```
BDOS                    =   0005H
FN_RETURN_LOGGED_IN_DRIVES  =   24
DRIVE_B_BIT_PATTERN     =   2
                ...
                ...
                LD   C,FN_RETURN_LOGGED_IN_DRIVES
                CALL BDOS

                LD   A,L                ;Check logged-in drives
                AND  DRIVE_B_BIT_PATTERN
                JP   Z,PROCESS          ;Jump if drive B
                                        ;logged-in

PROMPT:         ...                     ;Prompt operator to
                ...                     ;make drive available
                ...                     ;and log in drives by
                ...                     ;program
PROCESS:        ...
```

RETURN VERSION NUMBER 12

Registers on entry	*Registers on exit*
C: 0CH	**H:** System type **L:** Version number

Effect

Provides information that helps you write programs which can be used under different versions of CP/M.

When control passes back to your program, CP/M will have inserted in registers H and L two numbers that describe which type and version of the operating system your program is running under. The number in H describes the system type as shown in the left-hand table below. The number in L describes the version number as shown in the right-hand table below.

H	*System type*
0	CP/M
1	MP/M
2	CP/NET

L	*Version*
20	2.0
21	2.1
...	
2F	2.15
30	3.0
31	3.1
...	
3F	3.15

Example

The following example checks the system and version number and exits if the system is not CP/M.

```
BDOS            =   0005H
FN_VERSION_NO   =   12
                ...
                ...
                LD   C,FN_VERSION_NO
                CALL BDOS

                LD   A,H        ;Check system
                OR   A          ;type & version.
                JR   Z,PROCESS  ;Jump if CP/M
EXIT:           ...

PROCESS:        ...
```

17 SEARCH FOR FIRST ENTRY

Registers on entry	*Registers on exit*
C: 11H **DE:** FCB address	**A:** Return code

Effect

This function searches the directory for a match with the file given in the FCB addressed by DE. If the file is not found, the value FFH (255 decimal) is returned, otherwise a number in the range 0 to 3 is returned.

A wildcard facility is available: if you insert a ? character in any position in the filename, filetype or extent field, these positions will be ignored during the search. If the drive code field contains a ? character, the default disc will be searched. In this case, the function will return any matched entry belonging to any user number.

Note that you should not attempt any disc operation between Search for First Entry and Search for Next Entry.

Example

```
BDOS                    =    0005H
FN_SEARCH_FOR_FIRST     =    17
                  ...
                  ...
                  LD   C,FN_SEARCH_FOR_FIRST
                  CALL BDOS

                  INC  A
                  JN   Z,ENTRY_NOT_FOUND
PROCESS:          ...                    ;Process directory entry
                  ...
ENTRY_NOT_FOUND:  ...
```

SEARCH FOR NEXT ENTRY 18

Registers on entry	*Registers on exit*
C: 12H	A: Return code

Effect

This function is similar to Search for First Entry (function 17). However, in this case, the directory is scanned from the last matched entry. When no more directory items match, the value FFH (decimal 255) is returned. Return codes are the same as for Search for First Entry.

You should not attempt any disc operations between calls to Search for First and Search for Next.

Example

```
BDOS                    =    0005H
FN_SEARCH_FOR_NEXT      =    18
                  ...
                  ...
                  LD   C,FN_SEARCH_FOR_NEXT
                  CALL BDOS

                  INC  A
                  JN   Z,NO_MORE_ENTRIES
PROCESS:          ...                    ;Process this directory
                  ...                    ;entry
                  ...
NO_MORE_ENTRIES:  ...
```

14 SELECT DISC

Registers on entry	*Registers on exit*
C: 0EH E: Selected disc	

Effect

This function allows you to change the currently selected drive by program. You put the appropriate drive number in register E and then call the BDOS entry point. Drive numbers are in the range 0 to 15, where 0 refers to drive A and 15 to drive P.

The drive is logged-in until the next cold start, warm start (pressing CTRL/C) or call to Disc System Reset (function 13). If the disc in the drive is changed, the drive will go to a read-only status on the next disc operation to that drive.

FCBs that specify drive code zero refer to the currently selected drive; those that specify codes 1 to 16 refer to a specific drive (A to P).

Example

The following code selects drive B.

```
BDOS                =   0005H
FN_SELECT_DISC      =   14
                ...
                ...
                LD   E,1              ;Select drive B
                LD   C,FN_SELECT_DISC
                CALL BDOS

                ...
                ...
```

SET DMA ADDRESS 26

Registers on entry	*Registers on exit*
C: 1AH **DE:** New DMA address	

Effect

This function defines the start address of the disc data buffer for file operations (the DMA address). The disc data buffer itself will be 128 bytes in size.

The DMA address will be reset to 0080H when one of the following occurs.

- A cold start
- A warm start (pressing CTRL/C)
- Reset Disc System (function 13) is called

Example

```
BDOS                    =    0005H
FN_SET_DMA_ADDRESS      =    26
DATA_BUFFER:      ...

                  ...
                  ...
                  LD   DE,DATA_BUFFER
                  LD   C,FN_SET_DMA_ADDRESS
                  CALL BDOS
```

1AH

30 SET FILE ATTRIBUTES

Registers on entry	*Registers on exit*
C: 1EH **DE:** FCB address	**A:** Directory code

Effect

The *file attributes* are the most significant bits in bytes 1 to 11 of the FCB. They have the following significance.

M/S bit of byte	*Use*
1 to 4	Reserved for use by your programs
5 to 8	Reserved for future system use
9	Read-only attribute
10	System attribute
11	Reserved for future system use

If the Read-only attribute is set, the file will be protected against writing; if it is zero, you can write to the file.

If the System attribute is set, the file will not be listed by a DIR command.

To set an attribute, you first set the appropriate bit in the FCB and then use the Set Attribute function. The DE register pair should address an FCB with an unambiguous filename and the appropriate attributes should be set or unset.

If you want to check whether a file has any attributes set, you should use Search for First (function 17). This will return the file's directory entry in the form of an FCB and you can then look at the appropriate bit to see if the attribute is set or not.

You should not set attributes in any way other than by using Set Attributes.

Example

```
BDOS                    =   0005H
FN_SET_FILE_ATTRIBUTES  =   30
DATAFILE_FCB:   ...

                ...
                ...
                LD   DE,DATAFILE_FCB
                LD   C,FN_SET_FILE_ATTRIBUTES
                CALL BDOS
```

SET/GET USER CODE 32

Registers on entry	*Registers on exit*
C: 20H E: 0FFH (Get) User code (Set)	A: User code or no value

Effect

This function either changes the current user number or tells you what it is.

If register E contains FFH, the value of the current user number is returned in register A; it is in the range 0 to 15.

If register E contains any other value, the current user number is changed to the modulus 16 of this value (the remainder when the value is divided by 16).

Example

The following example sets the user code to 3.

```
BDOS                =   0005H
FN_SET_USER_CODE    =   32
USER_CODE           =   3
                    ...
                    ...
                    ...
                    LD   E,USER_CODE
                    LD   C,FN_SET_USER_CODE
                    CALL BDOS
```

The next example returns the user code and stops the program if it is not 1.

```
BDOS                =   0005H
FN_GET_USER_CODE    =   32
                    ...
                    ...
                    LD   E,255              ;Put FFH in E
                    LD   C,FN_GET_USER_CODE
                    CALL BDOS
                    DEC  A
                    JP   Z,PROCESS          ;User code = 1

EXIT:               ...
                    ...
PROCESS:            ...
```

8 SET I/O BYTE

Registers on entry	*Registers on exit*
C: 08H **E:** I/O byte value	

Effect

This function changes the IOBYTE value to that given in register E.

Example

```
BDOS            =   0005H
FN_SET_IO_BYTE  =   8
                ...
                ...
                LD  C,FN_SET_IO_BYTE
                LD  E,IO

                CALL BDOS
                ...
                ...
```

SET RANDOM SECTOR 36

Registers on entry	*On exit*
C: 24H **DE:** FCB address	Bytes 33 to 35 of FCB set to sector after end of file

Effect

When you are reading or writing a file sequentially, this function returns the random sector address of the current data sector in bytes 33 to 35 of the FCB addressed by register pair DE.

The information returned can be useful in two ways.

- You may scan a file sequentially to extract the positions of various fields. As each field is found, you can then call Set Random Sector to find the random sector position for the current sector. If the size of each data unit is 128 bytes, you can then put the sector position of each field in a table, together with the key of the field, for later retrieval. Once you have scanned the entire file and built up this table, you can move instantly to a specific sector by performing a random read.

  You can generalise the procedure to handle sectors containing variable-length records. Your program need store only the byte position relative to the start of the buffer, along with the field key and sector number, to find the exact start position of the field at a later time.
- If you want to process a file from an 'offset' position along the file, you can use Set Random Sector to help you. All you do is access the file sequentially until the offset is reached, use Set Random Sector to mark the position of the offset and then access the file with random read and write operations from that point

Example

```
BDOS                    =    0005H
FN_SET_RANDOM_SECTOR    =    36
DATAFILE_FCB       ...

                   ...
                   ...
                   LD   DE,DATAFILE_FCB
                   LD   C,FN_SET_RANDOM_SECTOR
                   CALL BDOS

                   ...       ;Random sector position is in
                             ;bytes 33 to 35 of DATAFILE_FCB.
```

24H

0 SYSTEM RESET

Registers on entry	*Registers on exit*
C: 00H	

Effect

Reloads the BDOS and CCP and then passes control to the CCP. The CCP re-initialises the disc system by selecting and logging in disc drive A. It will then select the drive previously selected by the CCP as the default drive.

This function has exactly the same effect as a jump to location BOOT.

Example

```
BDOS            =   0005H
FN_SYSTEM_RESET =   0
                ...
                ...
EXIT:           LD  C,FN_SYSTEM_RESET
                CALL BDOS
```

00H

WRITE PROTECT DISC 28

Registers on entry	*Registers on exit*
C: 1CH	

Effect

This function allows you to protect the currently selected drive until one of the following occurs.

- A cold or warm start (pressing CTRL/C)
- Reset Drive (function 37) is called
- Reset Disc System (function 13) is called

Any attempt to write to the disc will produce the message BDOS ERR on d: R/O.

Note that discs protected by Write Protect Disc are protected only against programs using CP/M BDOS functions. Programs that use BIOS or firmware routines can bypass this method of protection.

Example

```
BDOS                    =    0005H
FN_WRITE_PROTECT_DISC   =    28
                ...
                ...
                LD   C,FN_WRITE_PROTECT_DISC
                CALL BDOS
```

1CH

34 WRITE RANDOM

Registers on entry	*Registers on exit*
C: 22H **DE:** FCB address	**A:** Return code

Effect

This function writes a 128-byte sector of data to a file from the current data buffer. The file must have been opened using Open File (function 15).

The sector number in the file is specified by the contents of bytes 33 to 35 of the FCB. Byte 33 is the least-significant byte and byte 35 the most-significant byte. Bytes 33 and 34 together form a 16-bit sector number in the range 0 to 65535. Byte 35 is used by CP/M; you should set it to zero before using the function.

To use Write Random, you first set the sector number in bytes 33 and 34, set byte 35 to zero and then call the BDOS. Upon return, register A contains one of the error codes shown in Table D.3 (on page 132) or zero, if the operation was successful. The sector number in bytes 33 and 34 will not have been incremented, so a subsequent write will rewrite the same record.

CP/M automatically sets the logical extent and current sector values in the FCB. Thus, you can read or write the file sequentially from the current randomly accessed position. If you do this, you should note that the last sector that was randomly read will be reread when you switch to sequential reading, and the last sector will be rewritten when you switch to sequential writing.

Example

```
BDOS                =   0005H
FN_WRITE_RANDOM     =    34
DATAFILE_FCB    ...

                ...
                ...
                LD   (DATAFILE_FCB.RANDOM_BLOCK),HL ;Load random
                                                    ;block number
                                                    ;from HL
                LD   DE,DATAFILE_FCB
                LD   C,FN_WRITE_RANDOM
                CALL BDOS
                JP   Z,PROCESS              ;Write successful

ERROR:          ...                                    22H
                ...
PROCESS:        ...
```

WRITE RANDOM WITH ZERO FILL 40

Registers on entry	*Registers on exit*
C: 28H **DE:** FCB address	**A:** Return code

Effect

This function is similar to Write Random (function 34) with the exception that a previously unused sector is filled with zeros before the data is written.

Example

```
BDOS                        =    0005H
FN_WRITE_RANDOM_ZERO_FILL   =    40
DATAFILE_FCB:     ...

                  ...
                  ...
                  LD   DE,DATAFILE_FCB
                  LD   C,FN_WRITE_RANDOM_ZERO_FILL
                  CALL BDOS

                  JP   Z,PROCESS        ;No error

ERROR_IN_WRITE:   ...
                  ...

PROCESS:          ...
```

28H

21 WRITE SEQUENTIAL

Registers on entry	*Registers on exit*
C: 15H **DE:** FCB address	**A:** Return code

Effect

Writes a 128-byte sector of data from the current DMA address to the file. The file is described by the FCB whose address is held in DE and this FCB must have been activated by an Open File or Create File function.

If the write is successful, a zero value will be returned in register A, otherwise a non-zero value will be returned. This could happen, for example, if the disc is full.

The data sector is written to a position in the disc logical extent given by byte 32 of the FCB. This byte is then incremented by CP/M to point to the next sector. If byte 32 overflows, the next logical extent is opened and byte 32 reset to zero ready for the next write.

Write operations can take place on an existing file. In this case, new sectors will be appended to the file.

Example

```
BDOS                    =    0005H
FN_WRITE_SEQUENTIAL     =    21
DATAFILE_FCB:     ...

                  ...
                  ...
                  LD   DE,DATAFILE_FCB
                  LD   C,FN_WRITE_SEQUENTIAL
                  CALL BDOS
                  JP   Z,PROCESS
FAILED_WRITE:     ...                ;Disc full or hardware error
                  ...
PROCESS:          ...
```

Appendix E: BDOS Functions of CP/M 80 Family

FN	1.4	2.2	MP/M II	CP/NET 1.2	CP/M PLUS	PERSONAL CP/M	FN
0	SYSTEM RESET	SYSTEM RESET	SYSTEM RESET	SYSTEM RESET	SYSTEM RESET	SYSTEM RESET	0
1	READ CONSOLE	CONSOLE INPUT	CONSOLE INPUT	CONSOLE INPUT	CONSOLE INPUT	CONSOLE INPUT	1
2	WRITE CONSOLE	CONOUT	CONOUT	CONOUT	CONOUT	CONOUT	2
3	READER INPUT	READER INPUT	RAW CONSOLE INPUT	READER INPUT	AUXILIARY INPUT	AUXILIARY INPUT	3
4	PUNCH OUTPUT	PUNCH OUTPUT	RAW CONSOLE OUTPUT	PUNCH OUTPUT	AUXILIARY OUTPUT	AUXILIARY OUTPUT	4
5	LIST OUTPUT	LIST OUTPUT	LIST OUTPUT	LIST OUTPUT	LIST OUTPUT	LIST OUTPUT	5
6		DIRECT CONSOLE I/O	DIRECT CONSOLE I/O	DIRECT CONSOLE I/O	DIRECT CONSOLE I/O	DIRECT CONSOLE I/O	6
7	GET I/O STATUS	GET I/O BYTE	GET I/O BYTE	GET I/O BYTE	AUXILIARY STATUS	AUXILIARY I/P STATUS	7
8	SET I/O STATUS	SET I/O BYTE	SET I/O BYTE	SET I/O BYTE	AUXILIARY O/P STATUS	AUXILIARY O/P STATUS	8
9	PRINT CONTROL BUFFER	PRINT STRING	PRINT STRING	PRINT STRING	PRINT STRING	PRINT STRING	9
10	READ CONSOLE BUFFER	READ CONSOLE BUFFER	READ CONSOLE BUFFER	READ CONSOLE BUFFER	READ CONSOLE BUFFER	READ CONSOLE BUFFER	10
11	GET CONSOLE STATUS	GET CONSOLE STATUS	GET CONSOLE STATUS	GET CONSOLE STATUS	GET CONSOLE STATUS	GET CONSOLE STATUS	11
12	LIFT HEAD	RETURN VERSION NUMBER	RETURN VERSION NUMBER	RETURN VERSION NUMBER	RETURN VERSION NUMBER	RETURN VERSION NUMBER	12
13	INIT BDOS	RESET DISC SYSTEM	RESET DISC SYSTEM	RESET DISC SYSTEM	RESET DISC SYSTEM	RESET DISC SYSTEM	13
14	SELECT DISC	SELECT DISC	SELECT DISC	SELECT DISC	SELECT DISC	SELECT DISC	14
15	OPEN FILE	OPEN FILE	OPEN FILE	OPEN FILE	OPEN FILE	OPEN FILE	15
16	CLOSE FILE	CLOSE FILE	CLOSE FILE	CLOSE FILE	CLOSE FILE	CLOSE FILE	16
17	SEARCH FILE	SEARCH NEXT	SEARCH NEXT	SEARCH NEXT	SEARCH NEXT	SEARCH NEXT	17
18	SEARCH NEXT	SEARCH NEXT	SEARCH NEXT	SEARCH NEXT	SEARCH NEXT	SEARCH NEXT	18
19	DELETE FILE	DELETE FILE	DELETE FILE	DELETE FILE	DELETE FILE	DELETE FILE	19
20	READ SEQUENTIAL	READ SEQUENTIAL	READ SEQUENTIAL	READ SEQUENTIAL	READ SEQUENTIAL	READ SEQUENTIAL	20
21	WRITE SEQUENTIAL	WRITE SEQUENTIAL	WRITE SEQUENTIAL	WRITE SEQUENTIAL	WRITE SEQUENTIAL	WRITE SEQUENTIAL	21
22	CREATE FILE	CREATE FILE	CREATE FILE	CREATE FILE	CREATE FILE	CREATE FILE	22
23	RENAME FILE	RENAME FILE	RENAME FILE	RENAME FILE	RENAME FILE	RENAME FILE	23
24	RET LOGGED-IN DRIVES	RET LOGGED-IN DRIVES	RET LOGGED-IN DRIVES	RET LOGGED-IN DRIVES	RET LOGGED-IN DRIVES	RETURN CURRENT DISC	24
25	GET DRIVE NO	GET CURRENT DRIVE	GET CURRENT DRIVE	GET CURRENT DRIVE	RETURN CURRENT DISC	RETURN CURRENT DISC	25
26	SET DMA ADDRESS	SET DMA ADDRESS	SET DMA ADDRESS	SET DMA ADDRESS	SET DMA ADDRESS	SET DMA ADDRESS	26
27	GET ALLOC	GET ADD ALLOCN MAP	GET ADD ALLOCN MAP	GET ADD ALLOCN MAP	GET ADD ALLOCN MAP	GET ADD ALLOCN MAP	27
28		WRITE PROTECT DISC	WRITE PROTECT DISC	WRITE PROTECT DISC	WRITE PROTECT DISC	WRITE PROTECT DISC	28
29		GET R/O INDICATORS	GET R/O INDICATORS	GET R/O INDICATORS	GET R/O INDICATORS	GET R/O INDICATORS	29
30		SET FILE ATTRIBUTES	SET FILE ATTRIBUTES	SET FILE ATTRIBUTES	SET FILE ATTRIBUTES	SET FILE ATTRIBUTES	30
31		GET DISC PAR. ADD.	GET DISC PAR. ADD.	GET DISC PAR. ADD.	GET DISC PAR. ADD.	GET DISC PAR. ADD.	31
32		SET/GET USER CODE	SET/GET USER CODE	SET/GET USER CODE	SET/GET USER CODE	SET/GET USER CODE	32
33		READ RANDOM	READ RANDOM	READ RANDOM	READ RANDOM	READ RANDOM	33
34		WRITE RANDOM	WRITE RANDOM	WRITE RANDOM	WRITE RANDOM	WRITE RANDOM	34
35		COMPUTE FILE SIZE	COMPUTE FILE SIZE	COMPUTE FILE SIZE	COMPUTE FILE SIZE	COMPUTE FILE SIZE	35
36		SET RANDOM SECTOR	SET RANDOM SECTOR	SET RANDOM SECTOR	SET RANDOM SECTOR	SET RANDOM SECTOR	36
37		RESET DRIVE	RESET DRIVE	RESET DRIVE	RESET DRIVE	RESET DRIVE	37
38			ACCESS DRIVE	ACCESS DRIVE	ACCESS DRIVE		38
39			FREE DRIVE	FREE DRIVE	FREE DRIVE		39
40		WRITE RANDOM ZERO FILL	WRITE RANDOM ZERO FILL	WRITE RANDOM ZERO FILL	WRITE RANDOM ZERO FILL	WRITE RANDOM ZERO FILL	40
41			TEST & WRITE RECORD		TEST & WRITE RECORD		41
42			LOCK RECORD	LOCK RECORD	LOCK RECORD		42
43			UNLOCK RECORD	UNLOCK RECORD	UNLOCK RECORD		43
44			SET MULTI-SECTOR CT.		SET MULTI-SECTOR CT.		44
45			SET BDOS ERROR MODE	SET BDOS ERROR MODE	SET BDOS ERROR MODE	SET BDOS ERROR MODE	45
46			GET DISC FREE SPACE		GET DISC FREE SPACE		46
47			CHAIN TO PROGRAM		CHAIN TO PROGRAM		47
48			FLUSH BUFFERS		FLUSH BUFFERS	FLUSH BUFFERS	48
49					GET/SET SCB		49
50					DIRECT BIOS CALL		50
59					LOAD OVERLAY		59
60					CALL RSX		60
64				LOGIN			64
65				LOGOFF			65
66				SEND MSG ON NETWORK			66
67				GET MSG FROM NETWORK			67
68				GET NETWORK STATUS			68
69				GET CONFIG TABLE ADD			69

70		SET COMPATIBILITY ATT			70
71		GET SERVER CONFIG			71
98			FREE BLOCKS		98
99			TRUNCATE FILE		99
100	SET DIR LABEL		SET DIR LABEL		100
101	RETURN DIR LABEL		RETURN DIR LABEL		101
102	READ FILE XFCB		READ FILE DATE STAMPS AND PASSWORD MODE		102
103	WRITE FILE XFCB		WRITE FILE XFCB		103
104	SET DATE & TIME		SET DATE & TIME		104
105	GET DATE & TIME		GET DATE & TIME		105
106	SET DEFAULT P/WORD	SET DEFAULT P/WORD	SET DEFAULT P/WORD		106
107	RETURN SERIAL NO		RETURN SERIAL NO		107
108			GET/SET PROGRAM RETURN CODE		108
109			GET/SET CON MODE	GET/SET CON MODE	109
110			GET/SET O/P DELIMITERS	GET/SET O/P DELIMITERS	110
111			PRINT BLOCK	PRINT BLOCK	111
112			LIST BLOCK	LIST BLOCK	112
113				DIRECT SCREEN FNS	113
124				BYTE BLOCK COPY	124
125				BYTE BLOCK ALTER	125
128	ABSOLUTE MEMORY REQ				128
129	RELOC MEMORY REQ				129
130	MEMORY FREE				130
131	POLL				131
132	FLAG WAIT				132
133	FLAG SET				133
134	CREATE ONE				134
135	OPEN ONE				135
136	DELETE ONE				136
137	READ ONE				137
138	CONDITIONAL READ ONE				138
139	WRITE ONE				139
140	CONDITIONAL WRITE ONE				140
141	DELAY				141
142	DISPATCH				142
143	TERMINATE PROCESS				143
144	CREATE PROCESS				144
145	SET PRIORITY				145
146	ATTACH CONSOLE				146
147	DETACH CONSOLE				147
148	SET CONSOLE				148
149	ASSIGN CONSOLE				149
150	SEND CLI COMMAND				150
151	CALL RSP				151
152	PARSE FILENAME		PARSE FILENAME		152
153	GET CONSOLE NO				153
154	SYSTEM DATA ADDRESS				154
155	GET DATE & TIME				155
156	RETURN PROCESS DESCRIPTOR ADDRESS				156
157	ABORT PROCESS				157
158	ATTACH LIST				158
159	DETACH LIST				159
160	SET LIST				160
161	COND ATTACH LIST				161
162	COND ATTACH CON				162
163	RETURN MP/M VERSION NO				163
164	GET LIST NO				164

Appendix F: The ZASM Macro Assembler

Examples in this book are written using Research Machines Ltd's Z80 macro assembler ZASM.

The main differences between ZASM and Microsoft's M80 macro assembler, for example, are

- ZASM allows you to have longer symbol names and your programs can therefore be more readable than those generated by M80, which allows a maximum of only 6 significant characters
- ZASM is a true Z80 assembler using Zilog mnemonic conventions
- ZASM will generate the following types of file directly

  .COM files
  .HEX files
  .REL files

  M80 generates only .REL files

Index

Allocation block 67
  sizes 70
Allocation map 67
Appending to a file 43
Application program 4
ASCII Codes 107
Assembler 10

BASIC 88
Basic Disc Operating System 3, 106
Basic Input/Output System 3, 106
BDOS 3, 106
  bypassing 30
  function call 22, 109
  functions 22
Binary file 47
BIOS 3, 106
Blocking 46
BOOT 3
Boundary conditions 18
Breakpoints 77
Bugs 73

CCP 3, 4, 106
Check vectors 66
Close File 40, 111
Code, disassembling 80
Coding 8, 9
Command line 4
Command line tail 49
Compute File Size 89, 112
Conditional statements 19
Console buffer 27, 129
Console Command Processor 3, 4, 106
Console Input 26, 114
Console Output 24, 115
Console Status 28, 121
Coupling
  loose 19
  tight 18
CP/M 80 2, 156, 157
CP/NET 156, 157
Create File 39, 116
CTRL/Z 47
Currently selected drive 62

Data, raw 30
Data areas 47
DDT 76
De-blocking 46
Debugging a program 8, 14, 73
Default data buffer 40, 47
Default drive 62
Default FCB 50
Delete File 51, 117
Design 8, 16
Direct Console I/O 30, 118
Directory 36
Directory buffer 66
Directory operations 51
Disassembling code 80
Disc 36
  logical 69
Disc characteristics 64
Disc data buffer 40, 47
Disc data structures 64
Disc parameter block 47
Disc parameter header 65
Disc protection 63
Disc size 69
Disc write protection 63, 150
Displaying
  memory 78
  register set 78
DMA address 47
  setting 47, 144
Documentation (program) 8, 9
DPB 65, 68
  pointer 66
DPH 65
  pointer 65

Drive
  currently selected 62
  default 62
  logged-in 62
  logical 69
  status of 64, 124

Editor 10
End of file 47
Error handling
  with files 60
Extent 45, 82
  creating new 84
  opening 84
EXTERNAL 14, 89

FCB 40, 45
  default 50
FDOS 3
File 36
  appending 43
  binary 47
  closing 40, 111
  creating 39, 116
  deleting 51, 117
  end of 47
  opening 42, 126
  protection of 63
  reading 41
  renaming 51, 135
  size of 89, 112
  type of 40, 46
File attributes 63, 64, 145
Filename 40, 46
  passing via keyboard 49
Filetype 40, 46
Firmware 5
FSIZE 89, 96
Function code 22

Get
  Address of Allocation Map 68, 120
  Allocation Address 68, 120
  Console Status 28, 121
  Disc Parameters address 69, 122
  DPB address 69, 122
  I/O Byte 123
  Read-only Indicators 64, 124
  User Code 146
Global 14

High level languages and CP/M 6, 87

INCLUDE 13

Keyboard input 26, 28, 30, 114, 118

Librarian 13
Libraries 13
Link 13
Linker 13
List Output 24, 125
Listing file 12
Loading a program 8, 10
Logged-in drive 62
Logical drive 69
Loose coupling 19

Machine code 6
Machine code routine
  passing parameters 87
Make File 39, 116
Memory
  displaying 78
  usage 3, 106
MP/M 156, 157

Open File 42, 126
Operating system 1
Optimisation 8, 14

Page zero 20
PASCAL 89
Passing parameters 87
Patching
  data 79
  programs 80
Portability, program 19
Pre-allocated block map 69
Print String 24, 127
Printer output 24, 125
Program
  debugging 8, 14, 73
  documentation 8, 9
  loading 8, 10
  optimisation 8, 14
  portability 19
  running 8, 10
  source file 10
  testing 8, 14
Protecting discs and files 63
Punch Output 128

Random access 55, 56, 82
Raw data 30

Read Console Buffer 27, 129
Read Random 56, 131
Read Sequential 43, 133
Reader Input 134
Reading a file 4
Record 46
Record pointer 46, 56
Register set – displaying 78
REL file 13
Relocatable file 13
Rename File 51, 135
Reserved tracks 69
Reset
  Disc System 63, 64, 136
  Drive 64, 137
  System 22, 149
Return
  Current Disc 62, 138
  Current Drive 62, 138
  Logged-in Drives 63, 139
  Version Number 23, 140
Running a program 8, 10

Scratchpad area 66
Screen output 24, 30, 115, 118, 127
Search for First Entry 51, 141
Search for Next Entry 51, 142
Sector 45
Sector count 82
Sector translate table 65, 66
Sectors per track 69
Select Disc 62, 143
Sequential access 38, 82
Set
  DMA Address 47, 144
  File Attributes 63, 64, 145
  I/O Byte 147
  Random Sector 148
  User Code 146
Set/Get User Code 146
SID 76
Source code 10
Subroutine coupling 18
System parameters 3
System Reset 22, 149
System tables 106

TBASE 3
Testbed code 16
Testing a program 8, 14
Text editor 10
Text file 47
Tight coupling 18
TPA 3, 106
Tracks, reserved 69
Transient Program Area 3, 106

User programs 4
Utilities 2

Warm start 63
Wildcard characters 53, 141
Wildcards 53, 141
Write Protect Disc 63, 150
Write Random 58, 151
Write Random with Zero Fill 59, 152
Write Sequential 40, 82, 153

ZASM Macro Assembler 12, 158
ZSID 76