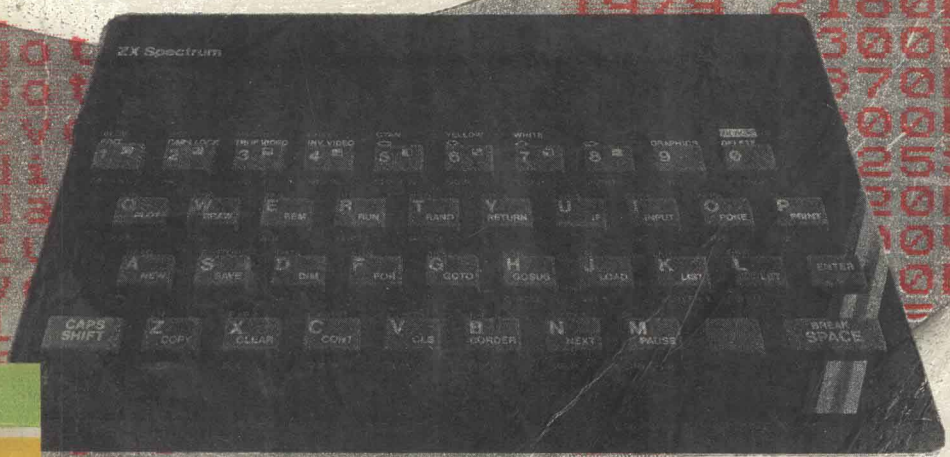
Information Handling for the ZX Spectrum

C.A. Street



Information Handling for the ZX Spectrum

C.A. STREET

McGRAW-HILL Book Company (UK) Limited

London · New York · St Louis · San Francisco · Auckland · Bogotá
Guatemala · Hamburg · Johannesburg · Lisbon · Madrid
Mexico · Montreal · New Delhi · Panama · Paris · San Juan · São
Paulo · Singapore · Sydney · Tokyo · Toronto

Published by
McGRAW-HILL Book Company (UK) Limited
MAIDENHEAD · BERKSHIRE · ENGLAND

British Library Cataloguing in Publication Data

Street, C.A.
Information handling for the ZX Spectrum.
1. Sinclair ZX Spectrum (computer)
I. Title
001.64'04 QA76.8.S625

ISBN 0-07-084707-X

Library of Congress Cataloging in Publication Data

Street, C.A.
Information handling for the ZX Spectrum.

1. Sinclair ZX Spectrum (Computer)–Programming. 2. Basic (Computer program language) 3. File organization (Computer science) I. Title. II. Title: Information handlig for the Z.X. Spectrum.
QA76.8.S627S84 1983 001.64'25 83-16252

ISBN 0-07-084707-X

1 2 3 4 5 CUP 8543

PRINTED AND BOUND IN GREAT BRITAIN BY CAMBRIDGE UNIVERSITY PRESS

INFORMATION HANDLING FOR THE ZX SPECTRUM

PREFACE

This book is intended for those who would like to make use of their computers in small-scale, but useful ways. Probably the most common application for large computers is that of storing, on a large scale, information of all types—from birth onwards our personal details, financial status, occupations, and other facts are recorded and used in large computer systems. By and large this activity improves the quality of life for the individual and lessens the burden of work (particularly its more routine aspects) in large public and private bodies. Suitably controlled and supervised, computerized data processing 'oils the wheels' of modern life with fewer sinister implications than are attached to many other twentieth century inventions.

Although the personal computer is slower and has less storage capacity than its larger relatives, it can still provide the same accuracy and convenience for home users. I have tried to provide two useful things in this volume. Firstly, the programs are working constructions which can be entered from keyboard or tape and used immediately. In the later part of the book, they have been designed so that parts of one may be used in another; thus once the various processes have been understood, the reader should be able to use modules from two or more individual programs in order to construct individually tailored systems to meet his own special needs. To the same end, I have endeavoured to make each module as flexible as possible, so changing a program designed for the storage of names and addresses to one for, say, a stamp catalogue should be a quick and easy operation.

Secondly, I have tried to use a 'structured' approach to planning the programs. The essence of this method is that the problem is thought through carefully before any coding—the act of writing BASIC statements themselves—is attempted. Having analysed the problem, a description of its methods is written in 'pseudo-code' which then leads on to the final BASIC version. The result should be programs which are easier to amend and understand. If the program is read alongside the pseudo-code description (the principles of this are introduced in the first chapter) its workings should be clear. I hope my readers will agree. The book is not, however, another text on structured programming—after the brief introduction in Chapter 1, it is used throughout the book in the course of writing programs which work and are useful. I hope that by writing in this manner I will have avoided the trap of artificial programs for the sake of

theoretically correct programming practice. I have learnt most of my programming skills in the course of trying to do something useful with a machine, in many cases by following and adapting programs produced by experts. It is my hope that my readers will gain similar benefits from working with the programs in this book. For maximum usefulness, a 48K ZX Spectrum is needed, but much is to be gained by adapting programs from one dialect of BASIC to another—a time consuming, but very instructive practice.

In Chapter 2 I have tried to tackle what is, to me, the most important and difficult part of the BASIC language. The IF . . . THEN construct is apparently simple, but has hidden traps, and I hope that a careful reading will help readers to recognize and avoid the pitfalls which await them. Thereafter the programs focus on the main objective: storing, retrieving, sorting and amending information in a standard 48K ZX Spectrum, using cassette tape for permanent storage. Many of the methods are related to those used in PROFILE, a file-handling program also published by McGraw-Hill, but a much larger book would be needed to give the full story of PROFILE, which was designed as a working tool, using some programming techniques beyond the scope of this book.

I would not, of course, claim that the programs are perfect, indeed I have deliberately refrained from using some of the features which can enliven any program. Colour and sound, for example, are used sparsely, but this was a calculated decision made to allow attention to be focused on the essentials of data-handling. A well constructed program can always have 'bells and whistles' added as an afterthought, but if the design starts with such matters, it may well not perform its basic functions efficiently.

Finally, I would like to thank all whose patient help has contributed so much. In particular I must mention Graham Bishop for his kind encouragement and, of course, my wife Irene without whose support neither PROFILE nor this book would have been possible.

CONTENTS

1 PROGRAMMING AND PLANNING

Programming a computer is a fulfilling activity. It presents a direct and compulsive challenge to our powers of organization, perseverance, and reasoning. The home computer is, of course, a many-faceted tool—with the right software it can entertain, inform, and act as an electronic filing cabinet—but it can also engage our wits and stamina in an attempt to make it carry out our particular will. This book is intended to help those who have learnt the elements of programming in BASIC and want to progress to a level at which their projects can be completed more quickly, work more efficiently, and are easier to amend if circumstances change.

Everyone has to begin by mastering the vocabulary of the computer's language and—more challenging—its syntax, or rules of grammar. One of the glories of Sinclair BASIC is its separation of the processes of syntax checking (carried out as soon as the ENTER key is pressed) and execution (after RUN). The 'syntax error' message which always manages to appear at the most interesting point on other machines during the run of a program is replaced by a friendly, if insistent, question mark which appears as soon as a line of BASIC is completed. I hope my readers will not mind if I digress further in praise of the version of BASIC used in this book—some of the points I make have been essential to the construction of the programs found here and to the associated PROFILE project. I hope the programs will be useful in at least two ways: they are meant to show some of the techniques for efficient storage and recall of data, but as tested and working programs they should, with the minimum of adaptation, fit many needs, particularly of the 'filing cabinet' variety.

A second unique feature of Spectrum BASIC is the 'keyword' entry system—ideal for the young and those whose typing, like my own, has never progressed far beyond the two-finger stage. More important, perhaps, is the fact that the keyboard becomes a dictionary of the language, albeit without definitions, so the irritating problem of having to look through the manual for a function which you know is somewhere (if only . . .) is almost eliminated. A by-product of keyword entry is that words are separated properly with spaces, as in any other piece of printed work. Readers of computer magazines will

know how confusing the condensed versions of programs sometimes found there can be:

```
  1ØFORK=STORE:PRINTED+K(S) :NEXTK
```

instead of

```
  1Ø FOR k=s TO re: PRINT ed+k(s) : NEXT k
```

in which not only does the separation between words help, but also the convention of

UPPER CASE for BASIC words
lower case for everything else

which I shall use throughout this book, with one exception. The only time for departure from this rule is when entering text between inverted commas in strings and PRINT statements, where capital letters are frequently desirable. The second rule used in producing versions of my programs for this book was also adopted for the sake of legibility. Spaces have been inserted wherever necessary in order to make them as easy to read and copy as possible. In particular, multiple statements after a line number have been separated so as to contain each statement on one screen line and to avoid the awkwardness of having words split between one line and the next. There is no reason for this other than for legibility; indeed, the programs run a little slower in expanded format, so if you copy them you may well wish to type in one continuous stream. In the later programs this would certainly be desirable as the extra spaces consume internal memory which is also used for the storage of data. All the programs have been run and tested on a Spectrum and printed out by the publishers on a dot-matrix printer direct from the working versions. If the reader wishes to save the trouble of typing the programs in himself (I intend no discrimination against programmers of the female sex), a cassette of all the major works in this book is obtainable direct from the publishers.

There are two other features of Sinclair BASIC which make it, in my view, the best current version of the beginner's language that I have used—which is not to say that it could not be improved still further. Neither aspect is immediately apparent, but with continued usage both have proved enormously useful. All dialects of the language store the values of variables in memory while a program is being run, but the Sinclair version is friendly to the extent that when a program is altered the values are not lost but simply, like the program, shifted around a little internally. Of course, if you then type RUN they are destroyed forever, but if GOTO is used instead, the old

values are retained and reused. For many applications this matters little, but when working with a mass of information which may have taken a long time to enter, this feature is invaluable. It extends to the storage of programs on cassette tape (SAVE outputs the program lines and variables to tape) and so, provided one is careful, important data need never be lost. This is particularly convenient when programs are being developed—data entered for test purposes can be kept and used over and over again almost as easily as if stored on a disc system.

The last characteristic of Sinclair BASIC I must praise is the way in which it has been kept as free as possible from rules which in essence say 'Don't'. As long as a statement has passed the test of the syntax checker, this dialect will do its utmost to produce what it thinks the user wants without taking too much for granted. Sometimes this will mean going to ridiculous lengths—try the following program to see what I mean:

```
10 LET f$ = "VAL$ f$" : PRINT VAL$ f$
```

The machine fills its entire internal memory while trying to work out this impossible problem! (It is a circular definition—'black is black, what is black?') As an aside, although many reviewers have found little use for the VAL$ function, I consider it one of the most elegant artefacts of the little interpreter and will show how it can be used in a later chapter.

1.1 Speed

There can be little argument that the Sinclair BASIC interpreter (the program which translates a set of BASIC statements into machine code and then carries out the resulting instructions) is slow compared with many others. For me, this is not a disturbing fact—it follows in part as a necessary consequence of some of the virtues described above. Other interpreters, for example, keep track of their variables by using sophisticated sets of pointers which cannot be updated every time a change to the program forces the variables to move internally. Pointer systems are very rapid, but relatively inflexible for certain purposes. Speed is nonetheless very useful at certain times in a program and I shall try to show how the necessary performance can be obtained when it is important. A few general rules might now be of use however:

1. Keep the number of GOSUBs and GOTOs to a minimum. Every time the interpreter meets one it has to search for the target, which takes time. This rule is particularly important if the GO

statement is in the middle of a repetitive process, where a delay of a few milliseconds each time the statement is met has to be multiplied by the number of times the process is repeated. This is an excellent rule for another reason. Programs with many jumps soon become difficult to understand because of their apparent complexity. I shall show in Chapter 2 how, in many cases, GO statements can be avoided altogether.

2. If the targets for GO statements can be placed near the beginning of a program, execution will be faster. This particularly applies to those line numbers which will be used frequently by other parts of the program.
3. Just as line numbers have to be found when required, so do variable values. If a number does not change during the execution of a program, it will help in terms of speed, if not memory usage, to use it as a number rather than a variable. For instance:

   10 LET z = 0
   20 IF p > z THEN . . .

   is marginally slower than

   20 IF p > 0 THEN . . .

   although, if the number zero is used a lot in the program, the first version will use less memory.
4. Plan programs carefully and try to break them down as far as possible into self-contained modules. A program which is allowed to grow like Topsy will most likely be an inefficient, rambling affair which duplicates some processes and does others in a less than purposeful (and therefore slow) manner.

1.2 Structured programming

Few issues of the popular computer magazines are allowed to pass without at least a mention of this topic, and rightly so, for within a short time of learning a computer language it dawns on most people that the process of drawing up the overall plan of a program is at least as important as the ability to code individual lines of BASIC. Amateur computer users have to be their own architects, draughtsmen, and bricklayers when building programs, and a structured approach is only an extension of sound practice. In many cases the early work is best done using a generalized language which is variously referred to as *program design language,* or *pseudo-code.* Which name you choose matters little, because what it amounts to is a 'half-way house' between ordinary language and almost any computer language of your choosing. In the rest of this chapter I shall introduce the main ideas of pseudo-code (I choose that name only because it is shorter), which will be used from time to time in the rest

of the book to design programs and parts thereof. Many purists would like a very close match between pseudo-code descriptions of programs and the actual program itself. Very few, if any, versions of BASIC allow this, most being like the curate's egg—good in parts. The Spectrum language is no exception and we shall find that there are two distinct jobs to do: first translate our thoughts into a pseudo-code description and then translate the result into BASIC.

1.3 Elements of pseudo-code

I do not intend to place myself in a strait-jacket in defining the type of pseudo-code descriptions I shall use in this book. It is enough to be faced with the strict rules of BASIC, and since we want a bridge between English and computer language which is for our use (not the computer's) I intend to be as flexible as possible. No one should feel constrained to follow my example—pseudo-code is a philosophy of planning, not another unyielding discipline.

BASIC words and functions—LET, THEN, AND, INT, . . .—will be freely used and are written in upper case. Variable names will be written in lower case, using a dollar sign where appropriate, but with no restriction on the number of characters involved. I shall often use underscore characters to join the individual words in a variable name together. The word PROC (for procedure) and a brief description will be used to head each section of code. For example:

```
PROC Electricity charge
// unit_chge in pence, fix_chge in pounds //
LET units_used = present_mter_read—prev_mter_read
LET cost_units = INT (units_used * unit_chge ) / 1ØØ
LET total = fix_chge + cost_units
ENDPROC
```

The section between the double '/' characters is a comment, and these will be included wherever they might seem to be useful. Note one of the most fundamental ideas in pseudo-code—where there is a beginning (PROC), there is also an end (ENDPROC). The same principle is used in the next example, which assumes a tax system in which a basic rate of 30 per cent is levied on the first £15 000 of taxable income (the remainder after allowances have been subtracted from the total salary) and a higher rate of 50 per cent on anything above that.

```
PROC Tax calculation
// Assume allowances have already been calculated //
```

```
LET tax_to_pay = Ø
LET taxable = salary − allows
IF taxable > 15ØØØ THEN
  LET taxable_high = taxable − 15ØØØ
  LET tax_to_pay = taxable_high * Ø.5
  LET taxable = 15ØØØ
ENDIF
LET tax_to_pay = tax_to_pay + taxable * Ø.3Ø
ENDPROC
```

Note the ENDIF which, together with identation, shows clearly the processes which have to be carried out if the condition is met and where to resume if not. In a BASIC program, this is best effected using a multiple statement line (line 13Ø, Fig. 1.1) without which it becomes difficult to avoid using GOTO statements. Apart from any considerations of speed, these make the program less easy to read. Note that, in line 14Ø, it is important to have the variable txp on both sides of the assignment, in order to cater for cases where the higher rate of calculation has been carried out.

```
 10 INPUT "saly";saly
 20 INPUT "alls";alls
100 REM taxcalc **************
110 LET txp=0
120 LET txbl=saly-alls
130 IF txbl>15000 THEN
      LET txblhi=txbl-15000:
       LET txp=txblhi*0.5:
        LET txbl=15000
140 LET txp=txp+txbl*0.3
150 PRINT "Tax to pay is ";txp
```

Figure 1.1

When an IF statement has to be used, the option of using an ELSE clause is tremendously valuable:

```
IF t$="Monday" THEN LET j$="Washing" ELSE LET j$="
Ironing"
```

Spectrum BASIC does not have this facility so we have to simply work around the problem. Sometimes this means using a GOTO statement:

```
1Ø IF d$ = "Monday" THEN LET j$ = "Washing" : GOTO 3Ø
2Ø LET j$="Ironing"
3Ø ...
```

but GOTO can often be avoided by using the 'logical' or Boolean functions (AND, OR, NOT) which I shall discuss at greater length in Chapter 2.

```
10 LET j$ = ("Washing" AND d$ = "Monday") +
                              ("Ironing" AND d$ <> "Monday")
```

In pseudo-code, I prefer to use indentation for ELSE clauses, as shown in the next example, which calculates a grade for a student from the marks gained in two assignments. The rules for doing so are stated in the comment lines at the beginning of the program. Even though ELSE clauses cannot be translated directly into Sinclair BASIC, they are most helpful in clarifying the way in which a process works.

```
PROC Assessment results
// Student is referred if either mark is less than 40% or average less
  than 50% //
// Pass for an average of 50% or more, and neither mark lower than
  40% //
// Merit for 65% or greater //
// Assume we have got two percentages, mark1 and mark2//
  IF mark1 < 40 OR mark2 < 40 THEN
  LET result$ = "Referred"
ELSE
  LET avge = (mark1 + mark2) /2
  IF avge < 50 THEN
    LET result$ = "Referred"
  ELSE
    IF avge >= 65 THEN
      LET result$ = "Merit"
    ELSE
      LET result$ = "Pass"
      ENDIF
 ENDIF
ENDIF
ENDPROC
```

The use of a single target line (200, Fig. 1.2) and the variable r$ makes the BASIC program (Fig. 1.2) much less complicated than it might otherwise have been. If the process of deciding an outcome is lengthy and the target line far away, some gain in readability can be made by using:

```
30 LET printline = 1200
```

and

11Ø . . .: GOTO printline, etc.

```
 10 INPUT "marks";m1,m2
100 REM Decide result *******
110 IF m1<40 OR m2<40 THEN
     LET r$="Referred":
      GO TO 200
120 LET avge=(m1+m2)/2
130 IF avge<50 THEN
     LET r$="Referred":
      GO TO 200
140 IF avge>=65 THEN
     LET r$="Merit": GO TO 200
150 LET r$="Pass"
200 PRINT "Result is ";r$
```

Figure 1.2

1.4 Case

In the preceding example, a decision between three possible grades was made using three IF statements. The CASE statement in pseudo-code allows a neat description of situations where many possibilities exist and allows us to avoid programs which have so many indentations that they begin to look like extremely agile snakes! It is usually reserved for dilemmas in which the course of action is determined by the value of one variable, as when throwing a die, but it is worth reminding ourselves that pseudo-code is an aid to good planning, not a fixed set of rules demanding strict adherence. The notation for this is shown below.

```
// Day of birth denoted by b$ //
CASE of b$
  b$ = "Monday"
    LET c$ = "fair of face"
  b$ = "Tuesday"
    LET c$ = "full of grace"
  b$ = "Wednesday"
    LET c$ = "full of woe"
  b$ = "Thursday"
    LET c$ = "far to go"
  b$ = "Friday"
    LET c$ = "loving and giving"
  b$ = "Saturday"
    LET c$ = "works hard for a living"
```

```
    b$ = "Sabbath day"
      LET c$ = "bonny and blythe and good and gay"
ENDCASE
```

Only one of the alternatives is to take effect. A blanket final option is allowable if none of the specific cases are met, and for this we use OTHERWISE. The use of this will be shown in the final program of this chapter, which is a simulation of the dice game, craps.

1.5 Loops

Almost all useful programs involve some sort of looping process. The statements inside a loop are repeated until some condition is satisfied, whereupon the program exits and (usually) carries on to the next statement after the loop. The most often used programming technique for looping is the FOR . . . NEXT method, but this is limiting, since the number of repetitions has to be given in the first statement and on many occasions it is simply not possible to predict in advance how many times the process will be repeated. I shall use two pseudo-code constructions to describe loops.

```
1. WHILE <condition>
     (statements to be repeated)
   ENDWHILE
2. REPEAT
     (statements to be repeated)
   ENDREPEAT ON <condition>
```

The only real difference between the two is the position of the condition to be tested. In the first it is at the beginning so the bracketed statements are not executed at all unless the condition is true at the start. The second always results in the loop being executed at least once because the condition is not tested until the very end. In many situations either can be used, most programmers having a preference for one type or the other. FOR . . . NEXT loops can usually be described either way; for example:

```
10 LET sum = 0
20 FOR c = 1 TO 8
30 LET sum = sum + c
40 NEXT c
50 PRINT "Total of first eight integers is "; sum
```